


NVIDIA Fortran CUDA Interfaces

Release 26.11ng-beta

NVIDIA Corporation

Nov 19, 2026

Contents

1 Fortran Interfaces and Wrappers **5**

2 Using CUDA Libraries from OpenACC Host Code **7**

3 Using CUDA Libraries from OpenACC Device Code **9**

4 Using CUDA Libraries from CUDA Fortran Host Code **11**

5 Using CUDA Libraries from CUDA Fortran Device Code **13**

6 Pointer Modes in cuBLAS and cuSPARSE **15**

7 Writing Your Own CUDA Interfaces **17**

8 NVIDIA Fortran Compiler Options **21**

9 BLAS Runtime APIs **23**
9.1 CUBLAS Definitions and Helper Functions . . . 24
9.1.1 cublasCreate . . . 25
9.1.2 cublasDestroy . . . 25
9.1.3 cublasGetVersion . . . 26
9.1.4 cublasSetStream . . . 26
9.1.5 cublasGetStream . . . 26
9.1.6 cublasGetStatusName . . . 26
9.1.7 cublasGetStatusString . . . 26
9.1.8 cublasGetPointerMode . . . 27
9.1.9 cublasSetPointerMode . . . 27
9.1.10 cublasGetAtomicsMode . . . 27
9.1.11 cublasSetAtomicsMode . . . 27
9.1.12 cublasGetMathMode . . . 28
9.1.13 cublasSetMathMode . . . 28
9.1.14 cublasGetSmCountTarget . . . 28
9.1.15 cublasSetSmCountTarget . . . 28
9.1.16 cublasGetHandle . . . 28
9.1.17 cublasSetVector . . . 29
9.1.18 cublasGetVector . . . 29
9.1.19 cublasSetMatrix . . . 29
9.1.20 cublasGetMatrix . . . 30
9.1.21 cublasSetVectorAsync . . . 30
9.1.22 cublasGetVectorAsync . . . 30
9.1.23 cublasSetMatrixAsync . . . 31
9.1.24 cublasGetMatrixAsync . . . 31
9.2 Single Precision Functions and Subroutines . . . 31
9.2.1 isamax . . . 31

9.2.2 isamin . . . 32
9.2.3 sasum . . . 32
9.2.4 saxpy . . . 33
9.2.5 scopy . . . 33
9.2.6 sdot . . . 34
9.2.7 snrm2 . . . 34
9.2.8 srot . . . 35
9.2.9 srotg . . . 35
9.2.10 srotm . . . 35
9.2.11 srotmg . . . 36
9.2.12 sscal . . . 36
9.2.13 sswap . . . 37
9.2.14 sgbmv . . . 37
9.2.15 sgemv . . . 38
9.2.16 sger . . . 38
9.2.17 ssbmv . . . 39
9.2.18 sspmv . . . 39
9.2.19 sspr . . . 40
9.2.20 sspr2 . . . 40
9.2.21 ssymv . . . 41
9.2.22 ssyr . . . 41
9.2.23 ssyr2 . . . 42
9.2.24 stbmv . . . 42
9.2.25 stbsv . . . 43
9.2.26 stpmv . . . 43
9.2.27 stpsv . . . 44
9.2.28 strmv . . . 44
9.2.29 strsv . . . 45
9.2.30 sgemm . . . 45
9.2.31 ssymm . . . 46
9.2.32 ssyrk . . . 46
9.2.33 ssyr2k . . . 47
9.2.34 ssyrkx . . . 48
9.2.35 strmm . . . 48
9.2.36 strsm . . . 49
9.2.37 cublasSgeam . . . 49
9.2.38 cublasSgemvBatched . . . 50
9.2.39 cublasSgemmBatched . . . 50
9.2.40 cublasSgelsBatched . . . 51
9.2.41 cublasSgeqrfBatched . . . 52
9.2.42 cublasSgetrfBatched . . . 52
9.2.43 cublasSgetriBatched . . . 52
9.2.44 cublasSgetrsBatched . . . 53
9.2.45 cublasSmatinvBatched . . . 53
9.2.46 cublasStrsmBatched . . . 53
9.2.47 cublasSgemvStridedBatched . . . 54
9.2.48 cublasSgemmStridedBatched . . . 55
9.3 Double Precision Functions and Subroutines . . . 56
9.3.1 idamax . . . 56
9.3.2 idamin . . . 56
9.3.3 dasum . . . 57
9.3.4 daxpy . . . 57
9.3.5 dcopy . . . 58
9.3.6 ddot . . . 58

9.3.7 dnrm2 . . . 58
9.3.8 drot . . . 59
9.3.9 drotg . . . 59
9.3.10 drotm . . . 60
9.3.11 drotmg . . . 60
9.3.12 dscal . . . 61
9.3.13 dswap . . . 61
9.3.14 dgbmv . . . 62
9.3.15 dgemv . . . 62
9.3.16 dger . . . 63
9.3.17 dsbmv . . . 63
9.3.18 dspmv . . . 64
9.3.19 dspr . . . 64
9.3.20 dspr2 . . . 65
9.3.21 dsymv . . . 65
9.3.22 dsyr . . . 66
9.3.23 dsyr2 . . . 66
9.3.24 dtbmv . . . 67
9.3.25 dtbsv . . . 67
9.3.26 dtpmv . . . 68
9.3.27 dtpsv . . . 68
9.3.28 dtrmv . . . 69
9.3.29 dtrsv . . . 69
9.3.30 dgemm . . . 70
9.3.31 dsymm . . . 70
9.3.32 dsyrk . . . 71
9.3.33 dsyr2k . . . 72
9.3.34 dsyrkx . . . 72
9.3.35 dtrmm . . . 73
9.3.36 dtrsm . . . 74
9.3.37 cublasDgeam . . . 74
9.3.38 cublasDgemvBatched . . . 75
9.3.39 cublasDgemmBatched . . . 75
9.3.40 cublasDgelsBatched . . . 76
9.3.41 cublasDgeqrfBatched . . . 77
9.3.42 cublasDgetrfBatched . . . 77
9.3.43 cublasDgetriBatched . . . 77
9.3.44 cublasDgetrsBatched . . . 78
9.3.45 cublasDmatinvBatched . . . 78
9.3.46 cublasDtrsmBatched . . . 78
9.3.47 cublasDgemvStridedBatched . . . 79
9.3.48 cublasDgemmStridedBatched . . . 80
9.4 Single Precision Complex Functions and Subroutines . . . 81
9.4.1 icamax . . . 81
9.4.2 icamin . . . 81
9.4.3 scasum . . . 82
9.4.4 caxpy . . . 82
9.4.5 ccopy . . . 83
9.4.6 cdotc . . . 83
9.4.7 cdotu . . . 83
9.4.8 scnrm2 . . . 84
9.4.9 crot . . . 84
9.4.10 csrot . . . 85
9.4.11 crotg . . . 85

9.4.12 cscal . . . 86
9.4.13 csscal . . . 86
9.4.14 cswap . . . 87
9.4.15 cgbmv . . . 87
9.4.16 cgemv . . . 88
9.4.17 cgerc . . . 88
9.4.18 cgeru . . . 89
9.4.19 csymv . . . 89
9.4.20 csyr . . . 90
9.4.21 csyr2 . . . 90
9.4.22 ctbmv . . . 91
9.4.23 ctbsv . . . 91
9.4.24 ctpmv . . . 92
9.4.25 ctpsv . . . 92
9.4.26 ctrmv . . . 93
9.4.27 ctrsv . . . 93
9.4.28 chbmv . . . 94
9.4.29 chemv . . . 94
9.4.30 chpmv . . . 95
9.4.31 cher . . . 95
9.4.32 cher2 . . . 96
9.4.33 chpr . . . 96
9.4.34 chpr2 . . . 97
9.4.35 cgemm . . . 97
9.4.36 csymm . . . 98
9.4.37 csyrk . . . 98
9.4.38 csyr2k . . . 99
9.4.39 csyrkx . . . 100
9.4.40 ctrmm . . . 100
9.4.41 ctrsm . . . 101
9.4.42 chemm . . . 101
9.4.43 cherk . . . 102
9.4.44 cher2k . . . 103
9.4.45 cherkx . . . 103
9.4.46 cublasCgeam . . . 104
9.4.47 cublasCgemvBatched . . . 104
9.4.48 cublasCgemmBatched . . . 105
9.4.49 cublasCgelsBatched . . . 106
9.4.50 cublasCgeqrfBatched . . . 106
9.4.51 cublasCgetrfBatched . . . 107
9.4.52 cublasCgetriBatched . . . 107
9.4.53 cublasCgetrsBatched . . . 107
9.4.54 cublasCmatinvBatched . . . 108
9.4.55 cublasCtrsmBatched . . . 108
9.4.56 cublasCgemvStridedBatched . . . 109
9.4.57 cublasCgemmStridedBatched . . . 109
9.5 Double Precision Complex Functions and Subroutines . . . 110
9.5.1 izamax . . . 110
9.5.2 izamin . . . 111
9.5.3 dzasum . . . 111
9.5.4 zaxpy . . . 112
9.5.5 zcopy . . . 112
9.5.6 zdotc . . . 113
9.5.7 zdotu . . . 113

9.5.8 dznrm2 . . . 114
9.5.9 zrot . . . 114
9.5.10 zsrot . . . 115
9.5.11 zrotg . . . 115
9.5.12 zscal . . . 115
9.5.13 zdscal . . . 116
9.5.14 zswap . . . 116
9.5.15 zgbmv . . . 117
9.5.16 zgemv . . . 117
9.5.17 zgerc . . . 118
9.5.18 zgeru . . . 118
9.5.19 zsymv . . . 119
9.5.20 zsyr . . . 119
9.5.21 zsyr2 . . . 120
9.5.22 ztbmv . . . 120
9.5.23 ztbsv . . . 121
9.5.24 ztpmv . . . 122
9.5.25 ztpsv . . . 122
9.5.26 ztrmv . . . 123
9.5.27 ztrsv . . . 123
9.5.28 zhbmv . . . 124
9.5.29 zhemv . . . 124
9.5.30 zhpmv . . . 125
9.5.31 zher . . . 125
9.5.32 zher2 . . . 126
9.5.33 zhpr . . . 126
9.5.34 zhpr2 . . . 127
9.5.35 zgemm . . . 127
9.5.36 zsymm . . . 128
9.5.37 zsyrk . . . 128
9.5.38 zsyr2k . . . 129
9.5.39 zsyrkx . . . 130
9.5.40 ztrmm . . . 130
9.5.41 ztrsm . . . 131
9.5.42 zhemm . . . 131
9.5.43 zherk . . . 132
9.5.44 zher2k . . . 133
9.5.45 zherkx . . . 133
9.5.46 cublasZgeam . . . 134
9.5.47 cublasZgemvBatched . . . 134
9.5.48 cublasZgemmBatched . . . 135
9.5.49 cublasZgelsBatched . . . 136
9.5.50 cublasZgeqrfBatched . . . 136
9.5.51 cublasZgetrfBatched . . . 137
9.5.52 cublasZgetriBatched . . . 137
9.5.53 cublasZgetrsBatched . . . 137
9.5.54 cublasZmatinvBatched . . . 138
9.5.55 cublasZtrsmBatched . . . 138
9.5.56 cublasZgemvStridedBatched . . . 139
9.5.57 cublasZgemmStridedBatched . . . 139
9.6 Half Precision Functions and Extension Functions . . . 140
9.6.1 cublasHgemvBatched . . . 141
9.6.2 cublasHgemvStridedBatched . . . 142
9.6.3 cublasHgemm . . . 143

9.6.4 cublasHgemmBatched . . . 144
9.6.5 cublasHgemmStridedBatched . . . 145
9.6.6 cublasIamaxEx . . . 146
9.6.7 cublasIaminEx . . . 146
9.6.8 cublasAsumEx . . . 146
9.6.9 cublasAxpyEx . . . 146
9.6.10 cublasCopyEx . . . 147
9.6.11 cublasDotEx . . . 147
9.6.12 cublasDotcEx . . . 148
9.6.13 cublasNrm2Ex . . . 148
9.6.14 cublasRotEx . . . 148
9.6.15 cublasRotgEx . . . 149
9.6.16 cublasRotmEx . . . 149
9.6.17 cublasRotmgEx . . . 149
9.6.18 cublasScalEx . . . 150
9.6.19 cublasSwapEx . . . 150
9.6.20 cublasGemmEx . . . 150
9.6.21 cublasGemmBatchedEx . . . 151
9.6.22 cublasGemmStridedBatchedEx . . . 151
9.7 CUBLAS V2 Module Functions . . . 152
9.7.1 Single Precision Functions and Subroutines . . . 152
9.7.1.1 isamax . . . 152
9.7.1.2 isamin . . . 153
9.7.1.3 sasum . . . 153
9.7.1.4 saxpy . . . 153
9.7.1.5 scopy . . . 153
9.7.1.6 sdot . . . 154
9.7.1.7 snrm2 . . . 154
9.7.1.8 srot . . . 154
9.7.1.9 srotg . . . 154
9.7.1.10 srotm . . . 154
9.7.1.11 srotmg . . . 155
9.7.1.12 sscal . . . 155
9.7.1.13 sswap . . . 155
9.7.1.14 sgbmv . . . 155
9.7.1.15 sgemv . . . 156
9.7.1.16 sger . . . 156
9.7.1.17 ssbmv . . . 156
9.7.1.18 sspmv . . . 156
9.7.1.19 sspr . . . 157
9.7.1.20 sspr2 . . . 157
9.7.1.21 ssymv . . . 157
9.7.1.22 ssyr . . . 157
9.7.1.23 ssyr2 . . . 158
9.7.1.24 stbmv . . . 158
9.7.1.25 stbsv . . . 158
9.7.1.26 stpmv . . . 158
9.7.1.27 stpsv . . . 159
9.7.1.28 strmv . . . 159
9.7.1.29 strsv . . . 159
9.7.1.30 sgemm . . . 159
9.7.1.31 ssymm . . . 160
9.7.1.32 ssyrk . . . 160
9.7.1.33 ssyr2k . . . 160

9.7.1.34 ssyrkx . . . 160
9.7.1.35 strmm . . . 161
9.7.1.36 strsm . . . 161
9.7.2 Double Precision Functions and Subroutines . . . 161
9.7.2.1 idamax . . . 161
9.7.2.2 idamin . . . 162
9.7.2.3 dasum . . . 162
9.7.2.4 daxpy . . . 162
9.7.2.5 dcopy . . . 162
9.7.2.6 ddot . . . 163
9.7.2.7 dnrm2 . . . 163
9.7.2.8 drot . . . 163
9.7.2.9 drotg . . . 163
9.7.2.10 drotm . . . 163
9.7.2.11 drotmg . . . 164
9.7.2.12 dscal . . . 164
9.7.2.13 dswap . . . 164
9.7.2.14 dgbmv . . . 164
9.7.2.15 dgemv . . . 165
9.7.2.16 dger . . . 165
9.7.2.17 dsbmv . . . 165
9.7.2.18 dspmv . . . 165
9.7.2.19 dspr . . . 166
9.7.2.20 dspr2 . . . 166
9.7.2.21 dsymv . . . 166
9.7.2.22 dsyr . . . 166
9.7.2.23 dsyr2 . . . 167
9.7.2.24 dtbmv . . . 167
9.7.2.25 dtbsv . . . 167
9.7.2.26 dtpmv . . . 167
9.7.2.27 dtpsv . . . 168
9.7.2.28 dtrmv . . . 168
9.7.2.29 dtrsv . . . 168
9.7.2.30 dgemm . . . 168
9.7.2.31 dsymm . . . 169
9.7.2.32 dsyrk . . . 169
9.7.2.33 dsyr2k . . . 169
9.7.2.34 dsyrkx . . . 169
9.7.2.35 dtrmm . . . 170
9.7.2.36 dtrsm . . . 170
9.7.3 Single Precision Complex Functions and Subroutines . . . 170
9.7.3.1 icamax . . . 170
9.7.3.2 icamin . . . 171
9.7.3.3 scasum . . . 171
9.7.3.4 caxpy . . . 171
9.7.3.5 ccopy . . . 171
9.7.3.6 cdotc . . . 172
9.7.3.7 cdotu . . . 172
9.7.3.8 scnrm2 . . . 172
9.7.3.9 crot . . . 172
9.7.3.10 csrot . . . 173
9.7.3.11 crotg . . . 173
9.7.3.12 cscal . . . 173
9.7.3.13 csscal . . . 173

9.7.3.14 cswap . . . 174
9.7.3.15 cgbmv . . . 174
9.7.3.16 cgemv . . . 174
9.7.3.17 cgerc . . . 174
9.7.3.18 cgeru . . . 175
9.7.3.19 csymv . . . 175
9.7.3.20 csyr . . . 175
9.7.3.21 csyr2 . . . 175
9.7.3.22 ctbmv . . . 176
9.7.3.23 ctbsv . . . 176
9.7.3.24 ctpmv . . . 176
9.7.3.25 ctpsv . . . 176
9.7.3.26 ctrmv . . . 177
9.7.3.27 ctrsv . . . 177
9.7.3.28 chbmv . . . 177
9.7.3.29 chemv . . . 177
9.7.3.30 chpmv . . . 178
9.7.3.31 cher . . . 178
9.7.3.32 cher2 . . . 178
9.7.3.33 chpr . . . 178
9.7.3.34 chpr2 . . . 179
9.7.3.35 cgemm . . . 179
9.7.3.36 csymm . . . 179
9.7.3.37 csyrk . . . 179
9.7.3.38 csyr2k . . . 180
9.7.3.39 csyrkx . . . 180
9.7.3.40 ctrmm . . . 180
9.7.3.41 ctrsm . . . 180
9.7.3.42 chemm . . . 181
9.7.3.43 cherk . . . 181
9.7.3.44 cher2k . . . 181
9.7.3.45 cherkx . . . 182
9.7.4 Double Precision Complex Functions and Subroutines . . . 182
9.7.4.1 izamax . . . 182
9.7.4.2 izamin . . . 182
9.7.4.3 dzasum . . . 183
9.7.4.4 zaxpy . . . 183
9.7.4.5 zcopy . . . 183
9.7.4.6 zdotc . . . 183
9.7.4.7 zdotu . . . 184
9.7.4.8 dznrm2 . . . 184
9.7.4.9 zrot . . . 184
9.7.4.10 zsrot . . . 184
9.7.4.11 zrotg . . . 185
9.7.4.12 zscal . . . 185
9.7.4.13 zdscal . . . 185
9.7.4.14 zswap . . . 185
9.7.4.15 zgbmv . . . 185
9.7.4.16 zgemv . . . 186
9.7.4.17 zgerc . . . 186
9.7.4.18 zgeru . . . 186
9.7.4.19 zsymv . . . 186
9.7.4.20 zsyr . . . 187
9.7.4.21 zsyr2 . . . 187

9.7.4.22 ztbmv . . . 187
9.7.4.23 ztbsv . . . 187
9.7.4.24 ztpmv . . . 188
9.7.4.25 ztpsv . . . 188
9.7.4.26 ztrmv . . . 188
9.7.4.27 ztrsv . . . 188
9.7.4.28 zhbmv . . . 189
9.7.4.29 zhemv . . . 189
9.7.4.30 zhpmv . . . 189
9.7.4.31 zher . . . 189
9.7.4.32 zher2 . . . 190
9.7.4.33 zhpr . . . 190
9.7.4.34 zhpr2 . . . 190
9.7.4.35 zgemm . . . 190
9.7.4.36 zsymm . . . 191
9.7.4.37 zsyrk . . . 191
9.7.4.38 zsyr2k . . . 191
9.7.4.39 zsyrkx . . . 191
9.7.4.40 ztrmm . . . 192
9.7.4.41 ztrsm . . . 192
9.7.4.42 zhemm . . . 192
9.7.4.43 zherk . . . 192
9.7.4.44 zher2k . . . 193
9.7.4.45 zherkx . . . 193
9.8 CUBLAS XT Module Functions . . . 193
9.8.1 cublasXtCreate . . . 195
9.8.2 cublasXtDestroy . . . 195
9.8.3 cublasXtDeviceSelect . . . 195
9.8.4 cublasXtSetBlockDim . . . 195
9.8.5 cublasXtGetBlockDim . . . 196
9.8.6 cublasXtSetCpuRoutine . . . 196
9.8.7 cublasXtSetCpuRatio . . . 196
9.8.8 cublasXtSetPinningMemMode . . . 196
9.8.9 cublasXtGetPinningMemMode . . . 197
9.8.10 cublasXtSgemm . . . 197
9.8.11 cublasXtSsymm . . . 197
9.8.12 cublasXtSsyrk . . . 198
9.8.13 cublasXtSsyr2k . . . 198
9.8.14 cublasXtSsyrkx . . . 198
9.8.15 cublasXtStrmm . . . 199
9.8.16 cublasXtStrsm . . . 199
9.8.17 cublasXtSspmm . . . 199
9.8.18 cublasXtCgemm . . . 200
9.8.19 cublasXtChemm . . . 200
9.8.20 cublasXtCherk . . . 200
9.8.21 cublasXtCher2k . . . 201
9.8.22 cublasXtCherkx . . . 201
9.8.23 cublasXtCsymm . . . 201
9.8.24 cublasXtCsyrk . . . 202
9.8.25 cublasXtCsyr2k . . . 202
9.8.26 cublasXtCsyrkx . . . 202
9.8.27 cublasXtCtrmm . . . 203
9.8.28 cublasXtCtrsm . . . 203
9.8.29 cublasXtCspmm . . . 203

9.8.30 cublasXtDgemm . . . 204
9.8.31 cublasXtDsymm . . . 204
9.8.32 cublasXtDsyrk . . . 204
9.8.33 cublasXtDsyr2k . . . 205
9.8.34 cublasXtDsyrkx . . . 205
9.8.35 cublasXtDtrmm . . . 205
9.8.36 cublasXtDtrsm . . . 206
9.8.37 cublasXtDspmm . . . 206
9.8.38 cublasXtZgemm . . . 206
9.8.39 cublasXtZhemm . . . 207
9.8.40 cublasXtZherk . . . 207
9.8.41 cublasXtZher2k . . . 207
9.8.42 cublasXtZherkx . . . 208
9.8.43 cublasXtZsymm . . . 208
9.8.44 cublasXtZsyrk . . . 208
9.8.45 cublasXtZsyr2k . . . 209
9.8.46 cublasXtZsyrkx . . . 209
9.8.47 cublasXtZtrmm . . . 209
9.8.48 cublasXtZtrsm . . . 210
9.8.49 cublasXtZspmm . . . 210
9.9 CUBLAS MP Module Functions . . . 210
9.9.1 cublasMpCreate . . . 212
9.9.2 cublasMpDestroy . . . 213
9.9.3 cublasMpSetStream . . . 213
9.9.4 cublasMpGetStream . . . 213
9.9.5 cublasMpSetEmulationStrategy . . . 213
9.9.6 cublasMpGetEmulationStrategy . . . 213
9.9.7 cublasMpSetFixedPointEmulationMaxMantissaBitCount . . . 214
9.9.8 cublasMpGetFixedPointEmulationMaxMantissaBitCount . . . 214
9.9.9 cublasMpSetFixedPointEmulationMantissaBitOffset . . . 214
9.9.10 cublasMpGetFixedPointEmulationMantissaBitOffset . . . 214
9.9.11 cublasMpSetFixedPointEmulationMantissaBitCountPointer . . . 215
9.9.12 cublasMpGetFixedPointEmulationMantissaBitCountPointer . . . 215
9.9.13 cublasMpSetFixedPointEmulationMantissaControl . . . 215
9.9.14 cublasMpGetFixedPointEmulationMantissaControl . . . 215
9.9.15 cublasMpSetEmulationSpecialValuesSupport . . . 216
9.9.16 cublasMpGetEmulationSpecialValuesSupport . . . 216
9.9.17 cublasMpGetVersion . . . 216
9.9.18 cublasMpGetStatusString . . . 216
9.9.19 cublasMpGridCreate . . . 216
9.9.20 cublasMpGridDestroy . . . 217
9.9.21 cublasMpBufferRegister . . . 217
9.9.22 cublasMpBufferDeregister . . . 217
9.9.23 cublasMpMalloc . . . 217
9.9.24 cublasMpFree . . . 218
9.9.25 cublasMpMatrixDescriptorCreate . . . 218
9.9.26 cublasMpMatrixDescriptorDestroy . . . 218
9.9.27 cublasMpMatrixDescriptorInit . . . 218
9.9.28 cublasMpNumroc . . . 219
9.9.29 cublasMpMatmulDescriptorCreate . . . 219
9.9.30 cublasMpMatmulDescriptorDestroy . . . 219
9.9.31 cublasMpMatmulDescriptorInit . . . 219
9.9.32 cublasMpMatmulDescriptorSetAttribute . . . 219
9.9.33 cublasMpMatmulDescriptorGetAttribute . . . 220

9.9.34 cublasMpGemr2D_bufferSize . . . 220
9.9.35 cublasMpGemr2D . . . 220
9.9.36 cublasMpTrmr2D_bufferSize . . . 221
9.9.37 cublasMpTrmr2D . . . 221
9.9.38 cublasMpGemm_bufferSize . . . 221
9.9.39 cublasMpGemm . . . 222
9.9.40 cublasMpMatmul_bufferSize . . . 222
9.9.41 cublasMpMatmul . . . 223
9.9.42 cublasMpSyrk_bufferSize . . . 223
9.9.43 cublasMpSyrk . . . 224
9.9.44 cublasMpSyr2k_bufferSize . . . 224
9.9.45 cublasMpSyr2k . . . 225
9.9.46 cublasMpSyrkx_bufferSize . . . 225
9.9.47 cublasMpSyrkx . . . 226
9.9.48 cublasMpSymm_bufferSize . . . 226
9.9.49 cublasMpSymm . . . 227
9.9.50 cublasMpTrsm_bufferSize . . . 227
9.9.51 cublasMpTrsm . . . 228
9.9.52 cublasMpTrmm_bufferSize . . . 228
9.9.53 cublasMpTrmm . . . 229
9.9.54 cublasMpGeadd_bufferSize . . . 229
9.9.55 cublasMpGeadd . . . 230
9.9.56 cublasMpTradd_bufferSize . . . 230
9.9.57 cublasMpTradd . . . 231
9.9.58 cublasMpHemm_bufferSize . . . 231
9.9.59 cublasMpHemm . . . 232
9.9.60 cublasMpHer2k_bufferSize . . . 232
9.9.61 cublasMpHer2k . . . 233
9.9.62 cublasMpHerkx_bufferSize . . . 233
9.9.63 cublasMpHerkx . . . 234
9.9.64 cublasMpHerk_bufferSize . . . 234
9.9.65 cublasMpHerk . . . 235
9.9.66 cublasMpLoggerSetFile . . . 235
9.9.67 cublasMpLoggerOpenFile . . . 235
9.9.68 cublasMpLoggerSetLevel . . . 236
9.9.69 cublasMpLoggerSetMask . . . 236
9.9.70 cublasMpLoggerForceDisable . . . 236
9.9.71 cublasMpLoggerSetCallback . . . 236

10 FFT Runtime Library APIs **237**
10.1 CUFFT Definitions and Helper Functions . . . 237
10.1.1 cufftSetStream . . . 238
10.1.2 cufftGetVersion . . . 239
10.1.3 cufftGetProperty . . . 239
10.1.4 cufftSetPlanPropertyInt64 . . . 239
10.1.5 cufftGetPlanPropertyInt64 . . . 239
10.1.6 cufftResetPlanProperty . . . 240
10.1.7 cufftSetAutoAllocation . . . 240
10.1.8 cufftSetWorkArea . . . 240
10.1.9 cufftDestroy . . . 240
10.2 CUFFT Plans and Estimated Size Functions . . . 241
10.2.1 cufftPlan1d . . . 241
10.2.2 cufftPlan2d . . . 241
10.2.3 cufftPlan3d . . . 241

10.2.4 cufftPlanMany . . . 242
10.2.5 cufftCreate . . . 242
10.2.6 cufftMakePlan1d . . . 242
10.2.7 cufftMakePlan2d . . . 243
10.2.8 cufftMakePlan3d . . . 243
10.2.9 cufftMakePlanMany . . . 243
10.2.10 cufftMakePlanMany64 . . . 244
10.2.11 cufftEstimate1d . . . 244
10.2.12 cufftEstimate2d . . . 244
10.2.13 cufftEstimate3d . . . 245
10.2.14 cufftEstimateMany . . . 245
10.2.15 cufftGetSize1d . . . 245
10.2.16 cufftGetSize2d . . . 245
10.2.17 cufftGetSize3d . . . 246
10.2.18 cufftGetSizeMany . . . 246
10.2.19 cufftGetSizeMany64 . . . 246
10.2.20 cufftGetSize . . . 247
10.3 CUFFT Execution Functions . . . 247
10.3.1 cufftExecC2C . . . 247
10.3.2 cufftExecR2C . . . 247
10.3.3 cufftExecC2R . . . 248
10.3.4 cufftExecZ2Z . . . 248
10.3.5 cufftExecD2Z . . . 248
10.3.6 cufftExecZ2D . . . 248
10.4 CUFFTXT Definitions and Helper Functions . . . 249
10.4.1 cufftXtSetGPUs . . . 251
10.4.2 cufftXtMalloc . . . 251
10.4.3 cufftXtFree . . . 251
10.4.4 cufftXtMemcpy . . . 252
10.5 CUFFTXT Plans and Work Area Functions . . . 252
10.5.1 cufftXtMakePlanMany . . . 252
10.5.2 cufftXtQueryPlan . . . 253
10.5.3 cufftXtSetWorkAreaPolicy . . . 253
10.5.4 cufftXtGetSizeMany . . . 253
10.5.5 cufftXtSetWorkArea . . . 254
10.5.6 cufftXtSetDistribution . . . 254
10.6 CUFFTXT Execution Functions . . . 254
10.6.1 cufftXtExec . . . 254
10.6.2 cufftXtExecDescriptor . . . 255
10.6.3 cufftXtExecDescriptorC2C . . . 255
10.6.4 cufftXtExecDescriptorZ2Z . . . 255
10.6.5 cufftXtExecDescriptorR2C . . . 255
10.6.6 cufftXtExecDescriptorD2Z . . . 256
10.6.7 cufftXtExecDescriptorC2R . . . 256
10.6.8 cufftXtExecDescriptorZ2D . . . 256
10.7 CUFFTXT Callback Functions . . . 256
10.7.1 cufftXtSetCallback . . . 256
10.7.2 cufftXtClearCallback . . . 257
10.7.3 cufftXtSetCallbackSharedSize . . . 257
10.7.4 cufftXtSetJITCallback . . . 257
10.8 CUFFTMP Functions . . . 257
10.8.1 cufftMpAttachComm . . . 258
10.8.2 cufftMpCreateReshape . . . 258
10.8.3 cufftMpAttachReshapeComm . . . 258

10.8.4 cufftMpGetReshapeSize . . . 258
10.8.5 cufftMpMakeReshape . . . 259
10.8.6 cufftMpExecReshapeAsync . . . 259
10.8.7 cufftMpDestroyReshape . . . 259

11 Random Number Runtime Library APIs **261**
11.1 CURAND Definitions and Helper Functions . . . 261
11.1.1 curandCreateGenerator . . . 263
11.1.2 curandCreateGeneratorHost . . . 263
11.1.3 curandDestroyGenerator . . . 263
11.1.4 curandGetVersion . . . 263
11.1.5 curandSetStream . . . 264
11.1.6 curandSetPseudoRandomGeneratorSeed . . . 264
11.1.7 curandSetGeneratorOffset . . . 264
11.1.8 curandSetGeneratorOrdering . . . 264
11.1.9 curandSetQuasiRandomGeneratorDimensions . . . 264
11.2 CURAND Generator Functions . . . 265
11.2.1 curandGenerate . . . 265
11.2.2 curandGenerateLongLong . . . 265
11.2.3 curandGenerateUniform . . . 265
11.2.4 curandGenerateUniformDouble . . . 265
11.2.5 curandGenerateNormal . . . 266
11.2.6 curandGenerateNormalDouble . . . 266
11.2.7 curandGeneratePoisson . . . 266
11.2.8 curandGenerateSeeds . . . 266
11.2.9 curandGenerateLogNormal . . . 267
11.2.10 curandGenerateLogNormalDouble . . . 267
11.3 CURAND Device Definitions and Functions . . . 267
11.3.1 curand_Init . . . 268
11.3.1.1 curandInitXORWOW . . . 269
11.3.1.2 curandInitMRG32k3a . . . 269
11.3.1.3 curandInitPhilox4_32_10 . . . 269
11.3.1.4 curandInitSobol32 . . . 269
11.3.1.5 curandInitScrambledSobol32 . . . 270
11.3.1.6 curandInitSobol64 . . . 270
11.3.1.7 curandInitScrambledSobol64 . . . 270
11.3.2 curand . . . 270
11.3.2.1 curandGetXORWOW . . . 271
11.3.2.2 curandGetMRG32k3a . . . 271
11.3.2.3 curandGetPhilox4_32_10 . . . 271
11.3.2.4 curandGetSobol32 . . . 271
11.3.2.5 curandGetScrambledSobol32 . . . 271
11.3.2.6 curandGetSobol64 . . . 272
11.3.2.7 curandGetScrambledSobol64 . . . 272
11.3.3 Curand_Normal . . . 272
11.3.3.1 curandNormalXORWOW . . . 272
11.3.3.2 curandNormalMRG32k3a . . . 272
11.3.3.3 curandNormalPhilox4_32_10 . . . 273
11.3.3.4 curandNormalSobol32 . . . 273
11.3.3.5 curandNormalScrambledSobol32 . . . 273
11.3.3.6 curandNormalSobol64 . . . 273
11.3.3.7 curandNormalScrambledSobol64 . . . 273
11.3.4 Curand_Normal_Double . . . 274
11.3.4.1 curandNormalDoubleXORWOW . . . 274

11.3.4.2 curandNormalDoubleMRG32k3a . . . 274
11.3.4.3 curandNormalDoublePhilox4_32_10 . . . 274
11.3.4.4 curandNormalDoubleSobol32 . . . 274
11.3.4.5 curandNormalDoubleScrambledSobol32 . . . 275
11.3.4.6 curandNormalDoubleSobol64 . . . 275
11.3.4.7 curandNormalDoubleScrambledSobol64 . . . 275
11.3.5 Curand_Log_Normal . . . 275
11.3.5.1 curandLogNormalXORWOW . . . 275
11.3.5.2 curandLogNormalMRG32k3a . . . 276
11.3.5.3 curandLogNormalPhilox4_32_10 . . . 276
11.3.5.4 curandLogNormalSobol32 . . . 276
11.3.5.5 curandLogNormalScrambledSobol32 . . . 276
11.3.5.6 curandLogNormalSobol64 . . . 276
11.3.5.7 curandLogNormalScrambledSobol64 . . . 277
11.3.6 Curand_Log_Normal_Double . . . 277
11.3.6.1 curandLogNormalDoubleXORWOW . . . 277
11.3.6.2 curandLogNormalDoubleMRG32k3a . . . 277
11.3.6.3 curandLogNormalDoublePhilox4_32_10 . . . 277
11.3.6.4 curandLogNormalDoubleSobol32 . . . 278
11.3.6.5 curandLogNormalDoubleScrambledSobol32 . . . 278
11.3.6.6 curandLogNormalDoubleSobol64 . . . 278
11.3.6.7 curandLogNormalDoubleScrambledSobol64 . . . 278
11.3.7 Curand_Uniform . . . 278
11.3.7.1 curandUniformXORWOW . . . 279
11.3.7.2 curandUniformMRG32k3a . . . 279
11.3.7.3 curandUniformPhilox4_32_10 . . . 279
11.3.7.4 curandUniformSobol32 . . . 279
11.3.7.5 curandUniformScrambledSobol32 . . . 279
11.3.7.6 curandUniformSobol64 . . . 280
11.3.7.7 curandUniformScrambledSobol64 . . . 280
11.3.8 Curand_Uniform_Double . . . 280
11.3.8.1 curandUniformDoubleXORWOW . . . 280
11.3.8.2 curandUniformDoubleMRG32k3a . . . 280
11.3.8.3 curandUniformDoublePhilox4_32_10 . . . 281
11.3.8.4 curandUniformDoubleSobol32 . . . 281
11.3.8.5 curandUniformDoubleScrambledSobol32 . . . 281
11.3.8.6 curandUniformDoubleSobol64 . . . 281
11.3.8.7 curandUniformDoubleScrambledSobol64 . . . 281

12 SPARSE Matrix Runtime Library APIs **283**
12.1 CUSPARSE Definitions and Helper Functions . . . 283
12.1.1 cusparseCreate . . . 289
12.1.2 cusparseDestroy . . . 289
12.1.3 cusparseGetErrorName . . . 289
12.1.4 cusparseGetErrorString . . . 289
12.1.5 cusparseGetVersion . . . 289
12.1.6 cusparseSetStream . . . 290
12.1.7 cusparseGetStream . . . 290
12.1.8 cusparseGetPointerMode . . . 290
12.1.9 cusparseSetPointerMode . . . 290
12.1.10 cusparseCreateMatDescr . . . 290
12.1.11 cusparseDestroyMatDescr . . . 291
12.1.12 cusparseSetMatType . . . 291
12.1.13 cusparseGetMatType . . . 291

12.1.14 cusparseSetMatFillMode . . . 291
12.1.15 cusparseGetMatFillMode . . . 291
12.1.16 cusparseSetMatDiagType . . . 292
12.1.17 cusparseGetMatDiagType . . . 292
12.1.18 cusparseSetMatIndexBase . . . 292
12.1.19 cusparseGetMatIndexBase . . . 292
12.1.20 cusparseCreateSolveAnalysisInfo . . . 292
12.1.21 cusparseDestroySolveAnalysisInfo . . . 293
12.1.22 cusparseGetLevelInfo . . . 293
12.1.23 cusparseCreateHybMat . . . 293
12.1.24 cusparseDestroyHybMat . . . 293
12.1.25 cusparseCreateCsrsv2Info . . . 293
12.1.26 cusparseDestroyCsrsv2Info . . . 294
12.1.27 cusparseCreateCsric02Info . . . 294
12.1.28 cusparseDestroyCsric02Info . . . 294
12.1.29 cusparseCreateCsrilu02Info . . . 294
12.1.30 cusparseDestroyCsrilu02Info . . . 294
12.1.31 cusparseCreateBsrsv2Info . . . 295
12.1.32 cusparseDestroyBsrsv2Info . . . 295
12.1.33 cusparseCreateBsric02Info . . . 295
12.1.34 cusparseDestroyBsric02Info . . . 295
12.1.35 cusparseCreateBsrilu02Info . . . 295
12.1.36 cusparseDestroyBsrilu02Info . . . 296
12.1.37 cusparseCreateBsrsm2Info . . . 296
12.1.38 cusparseDestroyBsrsm2Info . . . 296
12.1.39 cusparseCreateCsrgemm2Info . . . 296
12.1.40 cusparseDestroyCsrgemm2Info . . . 296
12.1.41 cusparseCreateColorInfo . . . 297
12.1.42 cusparseDestroyColorInfo . . . 297
12.1.43 cusparseCreateCsru2csrInfo . . . 297
12.1.44 cusparseDestroyCsru2csrInfo . . . 297
12.2 CUSPARSE Level 1 Functions . . . 297
12.2.1 cusparseSaxpyi . . . 297
12.2.2 cusparseDaxpyi . . . 298
12.2.3 cusparseCaxpyi . . . 298
12.2.4 cusparseZaxpyi . . . 298
12.2.5 cusparseSdoti . . . 299
12.2.6 cusparseDdoti . . . 299
12.2.7 cusparseCdoti . . . 299
12.2.8 cusparseZdoti . . . 300
12.2.9 cusparseCdotci . . . 300
12.2.10 cusparseZdotci . . . 300
12.2.11 cusparseSgthr . . . 301
12.2.12 cusparseDgthr . . . 301
12.2.13 cusparseCgthr . . . 301
12.2.14 cusparseZgthr . . . 302
12.2.15 cusparseSgthrz . . . 302
12.2.16 cusparseDgthrz . . . 302
12.2.17 cusparseCgthrz . . . 302
12.2.18 cusparseZgthrz . . . 303
12.2.19 cusparseSsctr . . . 303
12.2.20 cusparseDsctr . . . 303
12.2.21 cusparseCsctr . . . 304
12.2.22 cusparseZsctr . . . 304

12.2.23 cusparseSroti . . . 304
12.2.24 cusparseDroti . . . 304
12.3 CUSPARSE Level 2 Functions . . . 305
12.3.1 cusparseSbsrmv . . . 305
12.3.2 cusparseDbsrmv . . . 305
12.3.3 cusparseCbsrmv . . . 306
12.3.4 cusparseZbsrmv . . . 306
12.3.5 cusparseSbsrxmv . . . 307
12.3.6 cusparseDbsrxmv . . . 307
12.3.7 cusparseCbsrxmv . . . 308
12.3.8 cusparseZbsrxmv . . . 308
12.3.9 cusparseScsrmv . . . 309
12.3.10 cusparseDcsrmv . . . 309
12.3.11 cusparseCcsrmv . . . 310
12.3.12 cusparseZcsrmv . . . 310
12.3.13 cusparseScsrsv_analysis . . . 311
12.3.14 cusparseDcsrsv_analysis . . . 311
12.3.15 cusparseCcsrsv_analysis . . . 311
12.3.16 cusparseZcsrsv_analysis . . . 312
12.3.17 cusparseScsrsv_solve . . . 312
12.3.18 cusparseDcsrsv_solve . . . 312
12.3.19 cusparseCcsrsv_solve . . . 313
12.3.20 cusparseZcsrsv_solve . . . 313
12.3.21 cusparseSgemvi_bufferSize . . . 313
12.3.22 cusparseDgemvi_bufferSize . . . 314
12.3.23 cusparseCgemvi_bufferSize . . . 314
12.3.24 cusparseZgemvi_bufferSize . . . 314
12.3.25 cusparseSgemvi . . . 314
12.3.26 cusparseDgemvi . . . 315
12.3.27 cusparseCgemvi . . . 315
12.3.28 cusparseZgemvi . . . 315
12.3.29 cusparseShybmv . . . 316
12.3.30 cusparseDhybmv . . . 316
12.3.31 cusparseChybmv . . . 316
12.3.32 cusparseZhybmv . . . 317
12.3.33 cusparseShybsv_analysis . . . 317
12.3.34 cusparseDhybsv_analysis . . . 317
12.3.35 cusparseChybsv_analysis . . . 317
12.3.36 cusparseZhybsv_analysis . . . 318
12.3.37 cusparseShybsv_solve . . . 318
12.3.38 cusparseDhybsv_solve . . . 318
12.3.39 cusparseChybsv_solve . . . 318
12.3.40 cusparseZhybsv_solve . . . 319
12.3.41 cusparseSbsrsv2_bufferSize . . . 319
12.3.42 cusparseDbsrsv2_bufferSize . . . 319
12.3.43 cusparseCbsrsv2_bufferSize . . . 320
12.3.44 cusparseZbsrsv2_bufferSize . . . 320
12.3.45 cusparseSbsrsv2_analysis . . . 320
12.3.46 cusparseDbsrsv2_analysis . . . 321
12.3.47 cusparseCbsrsv2_analysis . . . 321
12.3.48 cusparseZbsrsv2_analysis . . . 321
12.3.49 cusparseSbsrsv2_solve . . . 322
12.3.50 cusparseDbsrsv2_solve . . . 322
12.3.51 cusparseCbsrsv2_solve . . . 323

12.3.52 cusparseZbsrsv2_solve . . . 323
12.3.53 cusparseXbsrsv2_zeroPivot . . . 323
12.3.54 cusparseScsrsv2_bufferSize . . . 324
12.3.55 cusparseDcsrsv2_bufferSize . . . 324
12.3.56 cusparseCcsrsv2_bufferSize . . . 324
12.3.57 cusparseZcsrsv2_bufferSize . . . 325
12.3.58 cusparseScsrsv2_analysis . . . 325
12.3.59 cusparseDcsrsv2_analysis . . . 325
12.3.60 cusparseCcsrsv2_analysis . . . 326
12.3.61 cusparseZcsrsv2_analysis . . . 326
12.3.62 cusparseScsrsv2_solve . . . 326
12.3.63 cusparseDcsrsv2_solve . . . 327
12.3.64 cusparseCcsrsv2_solve . . . 327
12.3.65 cusparseZcsrsv2_solve . . . 327
12.3.66 cusparseXcsrsv2_zeroPivot . . . 328
12.4 CUSPARSE Level 3 Functions . . . 328
12.4.1 cusparseScsrmm . . . 328
12.4.2 cusparseDcsrmm . . . 328
12.4.3 cusparseCcsrmm . . . 329
12.4.4 cusparseZcsrmm . . . 329
12.4.5 cusparseScsrmm2 . . . 330
12.4.6 cusparseDcsrmm2 . . . 330
12.4.7 cusparseCcsrmm2 . . . 330
12.4.8 cusparseZcsrmm2 . . . 331
12.4.9 cusparseScsrsm_analysis . . . 331
12.4.10 cusparseDcsrsm_analysis . . . 331
12.4.11 cusparseCcsrsm_analysis . . . 332
12.4.12 cusparseZcsrsm_analysis . . . 332
12.4.13 cusparseScsrsm_solve . . . 332
12.4.14 cusparseDcsrsm_solve . . . 333
12.4.15 cusparseCcsrsm_solve . . . 333
12.4.16 cusparseZcsrsm_solve . . . 333
12.4.17 cusparseScsrsm2_bufferSizeExt . . . 334
12.4.18 cusparseDcsrsm2_bufferSizeExt . . . 334
12.4.19 cusparseCcsrsm2_bufferSizeExt . . . 334
12.4.20 cusparseZcsrsm2_bufferSizeExt . . . 335
12.4.21 cusparseScsrsm2_analysis . . . 335
12.4.22 cusparseDcsrsm2_analysis . . . 335
12.4.23 cusparseCcsrsm2_analysis . . . 336
12.4.24 cusparseZcsrsm2_analysis . . . 336
12.4.25 cusparseScsrsm2_solve . . . 336
12.4.26 cusparseDcsrsm2_solve . . . 337
12.4.27 cusparseCcsrsm2_solve . . . 337
12.4.28 cusparseZcsrsm2_solve . . . 338
12.4.29 cusparseXcsrsm2_zeroPivot . . . 338
12.4.30 cusparseSbsrmm . . . 338
12.4.31 cusparseDbsrmm . . . 339
12.4.32 cusparseCbsrmm . . . 339
12.4.33 cusparseZbsrmm . . . 339
12.4.34 cusparseSbsrsm2_bufferSize . . . 340
12.4.35 cusparseDbsrsm2_bufferSize . . . 340
12.4.36 cusparseCbsrsm2_bufferSize . . . 340
12.4.37 cusparseZbsrsm2_bufferSize . . . 341
12.4.38 cusparseSbsrsm2_analysis . . . 341

12.4.39 cusparseDbsrsm2_analysis . . . 341
12.4.40 cusparseCbsrsm2_analysis . . . 342
12.4.41 cusparseZbsrsm2_analysis . . . 342
12.4.42 cusparseSbsrsm2_solve . . . 342
12.4.43 cusparseDbsrsm2_solve . . . 343
12.4.44 cusparseCbsrsm2_solve . . . 343
12.4.45 cusparseZbsrsm2_solve . . . 343
12.4.46 cusparseXbsrsm2_zeroPivot . . . 344
12.4.47 cusparseSgemmi . . . 344
12.4.48 cusparseDgemmi . . . 344
12.4.49 cusparseCgemmi . . . 345
12.4.50 cusparseZgemmi . . . 345
12.5 CUSPARSE Extra Functions . . . 346
12.5.1 cusparseXcsrgeamNnz . . . 346
12.5.2 cusparseScsrgeam . . . 346
12.5.3 cusparseDcsrgeam . . . 347
12.5.4 cusparseCcsrgeam . . . 347
12.5.5 cusparseZcsrgeam . . . 347
12.5.6 cusparseScsrgeam2_bufferSizeExt . . . 348
12.5.7 cusparseDcsrgeam2_bufferSizeExt . . . 348
12.5.8 cusparseCcsrgeam2_bufferSizeExt . . . 349
12.5.9 cusparseZcsrgeam2_bufferSizeExt . . . 349
12.5.10 cusparseXcsrgeam2Nnz . . . 350
12.5.11 cusparseScsrgeam2 . . . 350
12.5.12 cusparseDcsrgeam2 . . . 350
12.5.13 cusparseCcsrgeam2 . . . 351
12.5.14 cusparseZcsrgeam2 . . . 351
12.5.15 cusparseXcsrgemmNnz . . . 352
12.5.16 cusparseScsrgemm . . . 352
12.5.17 cusparseDcsrgemm . . . 352
12.5.18 cusparseCcsrgemm . . . 353
12.5.19 cusparseZcsrgemm . . . 353
12.5.20 cusparseScsrgemm2_bufferSizeExt . . . 354
12.5.21 cusparseDcsrgemm2_bufferSizeExt . . . 354
12.5.22 cusparseCcsrgemm2_bufferSizeExt . . . 354
12.5.23 cusparseZcsrgemm2_bufferSizeExt . . . 355
12.5.24 cusparseXcsrgemm2Nnz . . . 355
12.5.25 cusparseScsrgemm2 . . . 355
12.5.26 cusparseDcsrgemm2 . . . 356
12.5.27 cusparseCcsrgemm2 . . . 356
12.5.28 cusparseZcsrgemm2 . . . 357
12.6 CUSPARSE Preconditioning Functions . . . 357
12.6.1 cusparseScsric0 . . . 357
12.6.2 cusparseDcsric0 . . . 358
12.6.3 cusparseCcsric0 . . . 358
12.6.4 cusparseZcsric0 . . . 358
12.6.5 cusparseScsrilu0 . . . 359
12.6.6 cusparseDcsrilu0 . . . 359
12.6.7 cusparseCcsrilu0 . . . 359
12.6.8 cusparseZcsrilu0 . . . 360
12.6.9 cusparseSgtsv . . . 360
12.6.10 cusparseDgtsv . . . 360
12.6.11 cusparseCgtsv . . . 361
12.6.12 cusparseZgtsv . . . 361

12.6.13 cusparseSgtsv2_buffersize . . . 361
12.6.14 cusparseDgtsv2_buffersize . . . 362
12.6.15 cusparseCgtsv2_buffersize . . . 362
12.6.16 cusparseZgtsv2_buffersize . . . 362
12.6.17 cusparseSgtsv2 . . . 362
12.6.18 cusparseDgtsv2 . . . 363
12.6.19 cusparseCgtsv2 . . . 363
12.6.20 cusparseZgtsv2 . . . 363
12.6.21 cusparseSgtsv2_nopivot_buffersize . . . 364
12.6.22 cusparseDgtsv2_nopivot_buffersize . . . 364
12.6.23 cusparseCgtsv2_nopivot_buffersize . . . 364
12.6.24 cusparseZgtsv2_nopivot_buffersize . . . 364
12.6.25 cusparseSgtsv2_nopivot . . . 365
12.6.26 cusparseDgtsv2_nopivot . . . 365
12.6.27 cusparseCgtsv2_nopivot . . . 365
12.6.28 cusparseZgtsv2_nopivot . . . 366
12.6.29 cusparseSgtsv2StridedBatch_buffersize . . . 366
12.6.30 cusparseDgtsv2StridedBatch_buffersize . . . 366
12.6.31 cusparseCgtsv2StridedBatch_buffersize . . . 366
12.6.32 cusparseZgtsv2StridedBatch_buffersize . . . 367
12.6.33 cusparseSgtsv2StridedBatch . . . 367
12.6.34 cusparseDgtsv2StridedBatch . . . 367
12.6.35 cusparseCgtsv2StridedBatch . . . 368
12.6.36 cusparseZgtsv2StridedBatch . . . 368
12.6.37 cusparseSgtsvInterleavedBatch_buffersize . . . 368
12.6.38 cusparseDgtsvInterleavedBatch_buffersize . . . 369
12.6.39 cusparseCgtsvInterleavedBatch_buffersize . . . 369
12.6.40 cusparseZgtsvInterleavedBatch_buffersize . . . 369
12.6.41 cusparseSgtsvInterleavedBatch . . . 369
12.6.42 cusparseDgtsvInterleavedBatch . . . 370
12.6.43 cusparseCgtsvInterleavedBatch . . . 370
12.6.44 cusparseZgtsvInterleavedBatch . . . 371
12.6.45 cusparseSgpsvInterleavedBatch_buffersize . . . 371
12.6.46 cusparseDgpsvInterleavedBatch_buffersize . . . 371
12.6.47 cusparseCgpsvInterleavedBatch_buffersize . . . 372
12.6.48 cusparseZgpsvInterleavedBatch_buffersize . . . 372
12.6.49 cusparseSgpsvInterleavedBatch . . . 372
12.6.50 cusparseDgpsvInterleavedBatch . . . 373
12.6.51 cusparseCgpsvInterleavedBatch . . . 373
12.6.52 cusparseZgpsvInterleavedBatch . . . 373
12.6.53 cusparseScsric02_bufferSize . . . 374
12.6.54 cusparseDcsric02_bufferSize . . . 374
12.6.55 cusparseCcsric02_bufferSize . . . 374
12.6.56 cusparseZcsric02_bufferSize . . . 375
12.6.57 cusparseScsric02_analysis . . . 375
12.6.58 cusparseDcsric02_analysis . . . 375
12.6.59 cusparseCcsric02_analysis . . . 376
12.6.60 cusparseZcsric02_analysis . . . 376
12.6.61 cusparseScsric02 . . . 376
12.6.62 cusparseDcsric02 . . . 377
12.6.63 cusparseCcsric02 . . . 377
12.6.64 cusparseZcsric02 . . . 377
12.6.65 cusparseXcsric02_zeroPivot . . . 378
12.6.66 cusparseScsrilu02_numericBoost . . . 378

12.6.67 cusparseDcsrilu02_numericBoost . . . 378
12.6.68 cusparseCcsrilu02_numericBoost . . . 378
12.6.69 cusparseZcsrilu02_numericBoost . . . 379
12.6.70 cusparseScsrilu02_bufferSize . . . 379
12.6.71 cusparseDcsrilu02_bufferSize . . . 379
12.6.72 cusparseCcsrilu02_bufferSize . . . 380
12.6.73 cusparseZcsrilu02_bufferSize . . . 380
12.6.74 cusparseScsrilu02_analysis . . . 380
12.6.75 cusparseDcsrilu02_analysis . . . 381
12.6.76 cusparseCcsrilu02_analysis . . . 381
12.6.77 cusparseZcsrilu02_analysis . . . 381
12.6.78 cusparseScsrilu02 . . . 382
12.6.79 cusparseDcsrilu02 . . . 382
12.6.80 cusparseCcsrilu02 . . . 382
12.6.81 cusparseZcsrilu02 . . . 383
12.6.82 cusparseXcsrilu02_zeroPivot . . . 383
12.6.83 cusparseSbsric02_bufferSize . . . 383
12.6.84 cusparseDbsric02_bufferSize . . . 384
12.6.85 cusparseCbsric02_bufferSize . . . 384
12.6.86 cusparseZbsric02_bufferSize . . . 384
12.6.87 cusparseSbsric02_analysis . . . 385
12.6.88 cusparseDbsric02_analysis . . . 385
12.6.89 cusparseCbsric02_analysis . . . 385
12.6.90 cusparseZbsric02_analysis . . . 386
12.6.91 cusparseSbsric02 . . . 386
12.6.92 cusparseDbsric02 . . . 386
12.6.93 cusparseCbsric02 . . . 387
12.6.94 cusparseZbsric02 . . . 387
12.6.95 cusparseXbsric02_zeroPivot . . . 388
12.6.96 cusparseSbsrilu02_numericBoost . . . 388
12.6.97 cusparseDbsrilu02_numericBoost . . . 388
12.6.98 cusparseCbsrilu02_numericBoost . . . 388
12.6.99 cusparseZbsrilu02_numericBoost . . . 389
12.6.100 cusparseSbsrilu02_bufferSize . . . 389
12.6.101 cusparseDbsrilu02_bufferSize . . . 389
12.6.102 cusparseCbsrilu02_bufferSize . . . 390
12.6.103 cusparseZbsrilu02_bufferSize . . . 390
12.6.104 cusparseSbsrilu02_analysis . . . 390
12.6.105 cusparseDbsrilu02_analysis . . . 391
12.6.106 cusparseCbsrilu02_analysis . . . 391
12.6.107 cusparseZbsrilu02_analysis . . . 391
12.6.108 cusparseSbsrilu02 . . . 392
12.6.109 cusparseDbsrilu02 . . . 392
12.6.110 cusparseCbsrilu02 . . . 392
12.6.111 cusparseZbsrilu02 . . . 393
12.6.112 cusparseXbsrilu02_zeroPivot . . . 393
12.7 CUSPARSE Reordering Functions . . . 393
12.7.1 cusparseScsrColor . . . 394
12.7.2 cusparseDcsrColor . . . 394
12.7.3 cusparseCcsrColor . . . 394
12.7.4 cusparseZcsrColor . . . 395
12.8 CUSPARSE Format Conversion Functions . . . 395
12.8.1 cusparseSbsr2csr . . . 395
12.8.2 cusparseDbsr2csr . . . 395

12.8.3 cusparseCbsr2csr . . . 396
12.8.4 cusparseZbsr2csr . . . 396
12.8.5 cusparseXcoo2csr . . . 396
12.8.6 cusparseScsc2dense . . . 396
12.8.7 cusparseDcsc2dense . . . 397
12.8.8 cusparseCcsc2dense . . . 397
12.8.9 cusparseZcsc2dense . . . 397
12.8.10 cusparseScsc2hyb . . . 398
12.8.11 cusparseDcsc2hyb . . . 398
12.8.12 cusparseCcsc2hyb . . . 398
12.8.13 cusparseZcsc2hyb . . . 399
12.8.14 cusparseXcsr2bsrNnz . . . 399
12.8.15 cusparseScsr2bsr . . . 399
12.8.16 cusparseDcsr2bsr . . . 400
12.8.17 cusparseCcsr2bsr . . . 400
12.8.18 cusparseZcsr2bsr . . . 400
12.8.19 cusparseXcsr2coo . . . 401
12.8.20 cusparseScsr2csc . . . 401
12.8.21 cusparseDcsr2csc . . . 401
12.8.22 cusparseCcsr2csc . . . 401
12.8.23 cusparseZcsr2csc . . . 402
12.8.24 cusparseCsr2cscEx2_bufferSize . . . 402
12.8.25 cusparseCsr2cscEx2 . . . 402
12.8.26 cusparseScsr2dense . . . 403
12.8.27 cusparseDcsr2dense . . . 403
12.8.28 cusparseCcsr2dense . . . 403
12.8.29 cusparseZcsr2dense . . . 404
12.8.30 cusparseScsr2hyb . . . 404
12.8.31 cusparseDcsr2hyb . . . 404
12.8.32 cusparseCcsr2hyb . . . 405
12.8.33 cusparseZcsr2hyb . . . 405
12.8.34 cusparseSdense2csc . . . 405
12.8.35 cusparseDdense2csc . . . 406
12.8.36 cusparseCdense2csc . . . 406
12.8.37 cusparseZdense2csc . . . 406
12.8.38 cusparseSdense2csr . . . 406
12.8.39 cusparseDdense2csr . . . 407
12.8.40 cusparseCdense2csr . . . 407
12.8.41 cusparseZdense2csr . . . 407
12.8.42 cusparseSdense2hyb . . . 407
12.8.43 cusparseDdense2hyb . . . 408
12.8.44 cusparseCdense2hyb . . . 408
12.8.45 cusparseZdense2hyb . . . 408
12.8.46 cusparseShyb2csc . . . 408
12.8.47 cusparseDhyb2csc . . . 409
12.8.48 cusparseChyb2csc . . . 409
12.8.49 cusparseZhyb2csc . . . 409
12.8.50 cusparseShyb2csr . . . 410
12.8.51 cusparseDhyb2csr . . . 410
12.8.52 cusparseChyb2csr . . . 410
12.8.53 cusparseZhyb2csr . . . 410
12.8.54 cusparseShyb2dense . . . 411
12.8.55 cusparseDhyb2dense . . . 411
12.8.56 cusparseChyb2dense . . . 411

12.8.57 cusparseZhyb2dense . . . 411
12.8.58 cusparseSnnz . . . 412
12.8.59 cusparseDnnz . . . 412
12.8.60 cusparseCnnz . . . 412
12.8.61 cusparseZnnz . . . 413
12.8.62 cusparseSgebsr2gebsc_bufferSize . . . 413
12.8.63 cusparseDgebsr2gebsc_bufferSize . . . 413
12.8.64 cusparseCgebsr2gebsc_bufferSize . . . 414
12.8.65 cusparseZgebsr2gebsc_bufferSize . . . 414
12.8.66 cusparseSgebsr2gebsc . . . 414
12.8.67 cusparseDgebsr2gebsc . . . 415
12.8.68 cusparseCgebsr2gebsc . . . 415
12.8.69 cusparseZgebsr2gebsc . . . 415
12.8.70 cusparseSgebsr2gebsr_bufferSize . . . 416
12.8.71 cusparseDgebsr2gebsr_bufferSize . . . 416
12.8.72 cusparseCgebsr2gebsr_bufferSize . . . 416
12.8.73 cusparseZgebsr2gebsr_bufferSize . . . 417
12.8.74 cusparseXgebsr2gebsrNnz . . . 417
12.8.75 cusparseSgebsr2gebsr . . . 417
12.8.76 cusparseDgebsr2gebsr . . . 418
12.8.77 cusparseCgebsr2gebsr . . . 418
12.8.78 cusparseZgebsr2gebsr . . . 419
12.8.79 cusparseSgebsr2csr . . . 419
12.8.80 cusparseDgebsr2csr . . . 419
12.8.81 cusparseCgebsr2csr . . . 420
12.8.82 cusparseZgebsr2csr . . . 420
12.8.83 cusparseScsr2gebsr_bufferSize . . . 421
12.8.84 cusparseDcsr2gebsr_bufferSize . . . 421
12.8.85 cusparseCcsr2gebsr_bufferSize . . . 421
12.8.86 cusparseZcsr2gebsr_bufferSize . . . 421
12.8.87 cusparseXcsr2gebsrNnz . . . 422
12.8.88 cusparseScsr2gebsr . . . 422
12.8.89 cusparseDcsr2gebsr . . . 423
12.8.90 cusparseCcsr2gebsr . . . 423
12.8.91 cusparseZcsr2gebsr . . . 423
12.8.92 cusparseCreateIdentityPermutation . . . 424
12.8.93 cusparseXcoosort_bufferSize . . . 424
12.8.94 cusparseXcoosortByRow . . . 424
12.8.95 cusparseXcoosortByColumn . . . 425
12.8.96 cusparseXcsrsort_bufferSize . . . 425
12.8.97 cusparseXcsrsort . . . 425
12.8.98 cusparseXcscsort_bufferSize . . . 425
12.8.99 cusparseXcscsort . . . 426
12.8.100 cusparseScsru2csr_bufferSize . . . 426
12.8.101 cusparseDcsru2csr_bufferSize . . . 426
12.8.102 cusparseCcsru2csr_bufferSize . . . 426
12.8.103 cusparseZcsru2csr_bufferSize . . . 427
12.8.104 cusparseScsru2csr . . . 427
12.8.105 cusparseDcsru2csr . . . 427
12.8.106 cusparseCcsru2csr . . . 428
12.8.107 cusparseZcsru2csr . . . 428
12.8.108 cusparseScsr2csru . . . 428
12.8.109 cusparseDcsr2csru . . . 429
12.8.110 cusparseCcsr2csru . . . 429

12.8.111cusparseZcsr2csru 429
12.9 CUSPARSE Generic API Functions 430
12.9.1 cusparseDenseToSparse_bufferSize 430
12.9.2 cusparseDenseToSparse_analysis 430
12.9.3 cusparseDenseToSparse_convert 430
12.9.4 cusparseSparseToDense_bufferSize 431
12.9.5 cusparseSparseToDense 431
12.9.6 cusparseCreateSpVec 431
12.9.7 cusparseDestroySpVec 431
12.9.8 cusparseSpVecGet 432
12.9.9 cusparseSpVecGetIndexBase 432
12.9.10 cusparseSpVecGetValues 432
12.9.11 cusparseSpVecSetValues 432
12.9.12 cusparseCreateDnVec 433
12.9.13 cusparseDestroyDnVec 433
12.9.14 cusparseDnVecGet 433
12.9.15 cusparseDnVecGetValues 433
12.9.16 cusparseDnVecSetValues 434
12.9.17 cusparseCreateCoo 434
12.9.18 cusparseCreateCooAoS 434
12.9.19 cusparseCreateCsr 435
12.9.20 cusparseCreateBlockedEll 435
12.9.21 cusparseDestroySpMat 435
12.9.22 cusparseCooGet 435
12.9.23 cusparseCooAoSGet 436
12.9.24 cusparseCsrGet 436
12.9.25 cusparseBlockedEllGet 436
12.9.26 cusparseCsrSetPointers 437
12.9.27 cusparseCscSetPointers 437
12.9.28 cusparseSpMatGetFormat 437
12.9.29 cusparseSpMatGetIndexBase 437
12.9.30 cusparseSpMatGetSize 438
12.9.31 cusparseSpMatGetValues 438
12.9.32 cusparseSpMatSetValues 438
12.9.33 cusparseSpMatGetStridedBatch 438
12.9.34 cusparseSpMatSetStridedBatch 438
12.9.35 cusparseCooSetStridedBatch 439
12.9.36 cusparseCsrSetStridedBatch 439
12.9.37 cusparseBsrSetStridedBatch 439
12.9.38 cusparseSpMatGetAttribute 439
12.9.39 cusparseSpMatSetAttribute 440
12.9.40 cusparseCreateDnMat 440
12.9.41 cusparseDestroyDnMat 440
12.9.42 cusparseDnMatGet 440
12.9.43 cusparseDnMatGetValues 441
12.9.44 cusparseDnMatSetValues 441
12.9.45 cusparseDnMatGetStridedBatch 441
12.9.46 cusparseDnMatSetStridedBatch 441
12.9.47 cusparseSpVV_bufferSize 441
12.9.48 cusparseSpVV 442
12.9.49 cusparseSpMV_bufferSize 442
12.9.50 cusparseSpMV 442
12.9.51 cusparseSpSV_CreateDescr 443
12.9.52 cusparseSpSV_DestroyDescr 443

12.9.53 cusparseSpSV_bufferSize . . . 443
12.9.54 cusparseSpSV_analysis . . . 444
12.9.55 cusparseSpSV_solve . . . 444
12.9.56 cusparseSpSV_updateMatrix . . . 444
12.9.57 cusparseSpMM_bufferSize . . . 445
12.9.58 cusparseSpMM_preprocess . . . 445
12.9.59 cusparseSpMM . . . 445
12.9.60 cusparseSpSM_CreateDescr . . . 446
12.9.61 cusparseSpSM_DestroyDescr . . . 446
12.9.62 cusparseSpSM_bufferSize . . . 446
12.9.63 cusparseSpSM_analysis . . . 446
12.9.64 cusparseSpSM_solve . . . 447
12.9.65 cusparseSDDMM_bufferSize . . . 447
12.9.66 cusparseSDDMM_preprocess . . . 447
12.9.67 cusparseSDDMM . . . 448
12.9.68 cusparseSpGEMM_CreateDescr . . . 448
12.9.69 cusparseSpGEMM_DestroyDescr . . . 448
12.9.70 cusparseSpGEMM_workEstimation . . . 448
12.9.71 cusparseSpGEMM_getNumProducts . . . 449
12.9.72 cusparseSpGEMM_estimateMemory . . . 449
12.9.73 cusparseSpGEMM_compute . . . 449
12.9.74 cusparseSpGEMM_copy . . . 450
12.9.75 cusparseSpGEMMreuse_workEstimation . . . 451
12.9.76 cusparseSpGEMMreuse_nnz . . . 451
12.9.77 cusparseSpGEMMreuse_copy . . . 452
12.9.78 cusparseSpGEMMreuse_compute . . . 452

13 Matrix Solver Runtime Library APIs **453**
13.1 CUSOLVER Definitions and Helper Functions . . . 453
13.1.1 cusolverDnCreate . . . 456
13.1.2 cusolverDnDestroy . . . 456
13.1.3 cusolverDnCreateParams . . . 456
13.1.4 cusolverDnDestroyParams . . . 456
13.1.5 cusolverDnCreateSyevjInfo . . . 456
13.1.6 cusolverDnDestroySyevjInfo . . . 457
13.1.7 cusolverDnSetAdvOptions . . . 457
13.1.8 cusolverDnGetStream . . . 457
13.1.9 cusolverDnSetStream . . . 457
13.1.10 cusolverDnXsyevjSetTolerance . . . 457
13.1.11 cusolverDnXsyevjSetMaxSweeps . . . 458
13.1.12 cusolverDnXsyevjSetSortEig . . . 458
13.1.13 cusolverDnXsyevjGetResidual . . . 458
13.1.14 cusolverDnXsyevjGetSweeps . . . 458
13.2 cusolverDn Legacy API . . . 458
13.2.1 cusolverDnSpotrf_buffersize . . . 459
13.2.2 cusolverDnDpotrf_buffersize . . . 459
13.2.3 cusolverDnCpotrf_buffersize . . . 459
13.2.4 cusolverDnZpotrf_buffersize . . . 459
13.2.5 cusolverDnSpotrf . . . 460
13.2.6 cusolverDnDpotrf . . . 460
13.2.7 cusolverDnCpotrf . . . 460
13.2.8 cusolverDnZpotrf . . . 461
13.2.9 cusolverDnSpotrs . . . 461
13.2.10 cusolverDnDpotrs . . . 461

13.2.11 cusolverDnCpotrs . . . 462
13.2.12 cusolverDnZpotrs . . . 462
13.2.13 cusolverDnSpotrfBatched . . . 462
13.2.14 cusolverDnDpotrfBatched . . . 463
13.2.15 cusolverDnCpotrfBatched . . . 463
13.2.16 cusolverDnZpotrfBatched . . . 463
13.2.17 cusolverDnSpotrsBatched . . . 463
13.2.18 cusolverDnDpotrsBatched . . . 464
13.2.19 cusolverDnCpotrsBatched . . . 464
13.2.20 cusolverDnZpotrsBatched . . . 464
13.2.21 cusolverDnSpotri_buffersize . . . 465
13.2.22 cusolverDnDpotri_buffersize . . . 465
13.2.23 cusolverDnCpotri_buffersize . . . 465
13.2.24 cusolverDnZpotri_buffersize . . . 465
13.2.25 cusolverDnSpotri . . . 466
13.2.26 cusolverDnDpotri . . . 466
13.2.27 cusolverDnCpotri . . . 466
13.2.28 cusolverDnZpotri . . . 467
13.2.29 cusolverDnStrtri_buffersize . . . 467
13.2.30 cusolverDnDtrtri_buffersize . . . 467
13.2.31 cusolverDnCtrtri_buffersize . . . 468
13.2.32 cusolverDnZtrtri_buffersize . . . 468
13.2.33 cusolverDnStrtri . . . 468
13.2.34 cusolverDnDtrtri . . . 469
13.2.35 cusolverDnCtrtri . . . 469
13.2.36 cusolverDnZtrtri . . . 469
13.2.37 cusolverDnSlauum_buffersize . . . 470
13.2.38 cusolverDnDlauum_buffersize . . . 470
13.2.39 cusolverDnClauum_buffersize . . . 470
13.2.40 cusolverDnZlauum_buffersize . . . 470
13.2.41 cusolverDnSlauum . . . 471
13.2.42 cusolverDnDlauum . . . 471
13.2.43 cusolverDnClauum . . . 471
13.2.44 cusolverDnZlauum . . . 472
13.2.45 cusolverDnSgetrf_buffersize . . . 472
13.2.46 cusolverDnDgetrf_buffersize . . . 472
13.2.47 cusolverDnCgetrf_buffersize . . . 472
13.2.48 cusolverDnZgetrf_buffersize . . . 473
13.2.49 cusolverDnSgetrf . . . 473
13.2.50 cusolverDnDgetrf . . . 473
13.2.51 cusolverDnCgetrf . . . 473
13.2.52 cusolverDnZgetrf . . . 474
13.2.53 cusolverDnSgetrs . . . 474
13.2.54 cusolverDnDgetrs . . . 474
13.2.55 cusolverDnCgetrs . . . 475
13.2.56 cusolverDnZgetrs . . . 475
13.2.57 cusolverDnSlaswp . . . 475
13.2.58 cusolverDnDlaswp . . . 476
13.2.59 cusolverDnClaswp . . . 476
13.2.60 cusolverDnZlaswp . . . 476
13.2.61 cusolverDnSgeqrf_buffersize . . . 476
13.2.62 cusolverDnDgeqrf_buffersize . . . 477
13.2.63 cusolverDnCgeqrf_buffersize . . . 477
13.2.64 cusolverDnZgeqrf_buffersize . . . 477

13.2.65 cusolverDnSgeqrf . . . 477
13.2.66 cusolverDnDgeqrf . . . 478
13.2.67 cusolverDnCgeqrf . . . 478
13.2.68 cusolverDnZgeqrf . . . 478
13.2.69 cusolverDnSorgqr_buffersize . . . 478
13.2.70 cusolverDnDorgqr_buffersize . . . 479
13.2.71 cusolverDnCorgqr_buffersize . . . 479
13.2.72 cusolverDnZorgqr_buffersize . . . 479
13.2.73 cusolverDnSorgqr . . . 480
13.2.74 cusolverDnDorgqr . . . 480
13.2.75 cusolverDnCorgqr . . . 480
13.2.76 cusolverDnZorgqr . . . 480
13.2.77 cusolverDnSormqr_buffersize . . . 481
13.2.78 cusolverDnDormqr_buffersize . . . 481
13.2.79 cusolverDnCormqr_buffersize . . . 481
13.2.80 cusolverDnZormqr_buffersize . . . 482
13.2.81 cusolverDnSormqr . . . 482
13.2.82 cusolverDnDormqr . . . 482
13.2.83 cusolverDnCormqr . . . 483
13.2.84 cusolverDnZormqr . . . 483
13.2.85 cusolverDnSsytrf_buffersize . . . 483
13.2.86 cusolverDnDsytrf_buffersize . . . 484
13.2.87 cusolverDnCsytrf_buffersize . . . 484
13.2.88 cusolverDnZsytrf_buffersize . . . 484
13.2.89 cusolverDnSsytrf . . . 484
13.2.90 cusolverDnDsytrf . . . 485
13.2.91 cusolverDnCsytrf . . . 485
13.2.92 cusolverDnZsytrf . . . 485
13.2.93 cusolverDnSsytrs_bufferSize . . . 486
13.2.94 cusolverDnDsytrs_bufferSize . . . 486
13.2.95 cusolverDnCsytrs_bufferSize . . . 486
13.2.96 cusolverDnZsytrs_bufferSize . . . 487
13.2.97 cusolverDnSsytrs . . . 487
13.2.98 cusolverDnDsytrs . . . 487
13.2.99 cusolverDnCsytrs . . . 488
13.2.100 cusolverDnZsytrs . . . 488
13.2.101 cusolverDnSsytri_bufferSize . . . 488
13.2.102 cusolverDnDsytri_bufferSize . . . 489
13.2.103 cusolverDnCsytri_bufferSize . . . 489
13.2.104 cusolverDnZsytri_bufferSize . . . 489
13.2.105 cusolverDnSsytri . . . 490
13.2.106 cusolverDnDsytri . . . 490
13.2.107 cusolverDnCsytri . . . 490
13.2.108 cusolverDnZsytri . . . 491
13.3 cusolverDn Legacy Eigenvalue Solver API . . . 491
13.3.1 cusolverDnSgebrd_buffersize . . . 491
13.3.2 cusolverDnDgebrd_buffersize . . . 491
13.3.3 cusolverDnCgebrd_buffersize . . . 492
13.3.4 cusolverDnZgebrd_buffersize . . . 492
13.3.5 cusolverDnSgebrd . . . 492
13.3.6 cusolverDnDgebrd . . . 492
13.3.7 cusolverDnCgebrd . . . 493
13.3.8 cusolverDnZgebrd . . . 493
13.3.9 cusolverDnSorgbr_buffersize . . . 494

13.3.10 cusolverDnDorgbr_buffersize . . . 494
13.3.11 cusolverDnCungbr_buffersize . . . 494
13.3.12 cusolverDnZungbr_buffersize . . . 495
13.3.13 cusolverDnSorgbr . . . 495
13.3.14 cusolverDnDorgbr . . . 495
13.3.15 cusolverDnCungbr . . . 496
13.3.16 cusolverDnZungbr . . . 496
13.3.17 cusolverDnSsytrd_buffersize . . . 496
13.3.18 cusolverDnDsytrd_buffersize . . . 497
13.3.19 cusolverDnChetrd_buffersize . . . 497
13.3.20 cusolverDnZhetrd_buffersize . . . 497
13.3.21 cusolverDnSsytrd . . . 498
13.3.22 cusolverDnDsytrd . . . 498
13.3.23 cusolverDnChetrd . . . 498
13.3.24 cusolverDnZhetrd . . . 499
13.3.25 cusolverDnSormtr_buffersize . . . 499
13.3.26 cusolverDnDormtr_buffersize . . . 499
13.3.27 cusolverDnCunmtr_buffersize . . . 500
13.3.28 cusolverDnZunmtr_buffersize . . . 500
13.3.29 cusolverDnSormtr . . . 500
13.3.30 cusolverDnDormtr . . . 501
13.3.31 cusolverDnCunmtr . . . 501
13.3.32 cusolverDnZunmtr . . . 501
13.3.33 cusolverDnSorgtr_buffersize . . . 502
13.3.34 cusolverDnDorgtr_buffersize . . . 502
13.3.35 cusolverDnCungtr_buffersize . . . 502
13.3.36 cusolverDnZungtr_buffersize . . . 503
13.3.37 cusolverDnSorgtr . . . 503
13.3.38 cusolverDnDorgtr . . . 503
13.3.39 cusolverDnCungtr . . . 504
13.3.40 cusolverDnZungtr . . . 504
13.3.41 cusolverDnSgesvd_buffersize . . . 504
13.3.42 cusolverDnDgesvd_buffersize . . . 505
13.3.43 cusolverDnCgesvd_buffersize . . . 505
13.3.44 cusolverDnZgesvd_buffersize . . . 505
13.3.45 cusolverDnSgesvd . . . 505
13.3.46 cusolverDnDgesvd . . . 506
13.3.47 cusolverDnCgesvd . . . 506
13.3.48 cusolverDnZgesvd . . . 506
13.3.49 cusolverDnSsyevd_buffersize . . . 507
13.3.50 cusolverDnDsyevd_buffersize . . . 507
13.3.51 cusolverDnCheevd_buffersize . . . 507
13.3.52 cusolverDnZheevd_buffersize . . . 508
13.3.53 cusolverDnSsyevd . . . 508
13.3.54 cusolverDnDsyevd . . . 508
13.3.55 cusolverDnCheevd . . . 509
13.3.56 cusolverDnZheevd . . . 509
13.3.57 cusolverDnSsyevdx_buffersize . . . 509
13.3.58 cusolverDnDsyevdx_buffersize . . . 510
13.3.59 cusolverDnCheevdx_buffersize . . . 510
13.3.60 cusolverDnZheevdx_buffersize . . . 510
13.3.61 cusolverDnSsyevdx . . . 511
13.3.62 cusolverDnDsyevdx . . . 511
13.3.63 cusolverDnCheevdx . . . 511

13.3.64 cusolverDnZheevdx . . . 512
13.3.65 cusolverDnSsyevj_buffersize . . . 512
13.3.66 cusolverDnDsyevj_buffersize . . . 512
13.3.67 cusolverDnCheevj_buffersize . . . 513
13.3.68 cusolverDnZheevj_buffersize . . . 513
13.3.69 cusolverDnSsyevj . . . 513
13.3.70 cusolverDnDsyevj . . . 514
13.3.71 cusolverDnCheevj . . . 514
13.3.72 cusolverDnZheevj . . . 514
13.3.73 cusolverDnSsyevjBatched_bufferSize . . . 515
13.3.74 cusolverDnDsyevjBatched_bufferSize . . . 515
13.3.75 cusolverDnCheevjBatched_bufferSize . . . 515
13.3.76 cusolverDnZheevjBatched_bufferSize . . . 516
13.3.77 cusolverDnSsyevjBatched . . . 516
13.3.78 cusolverDnDsyevjBatched . . . 516
13.3.79 cusolverDnCheevjBatched . . . 517
13.3.80 cusolverDnZheevjBatched . . . 517
13.3.81 cusolverDnSsygvd_buffersize . . . 517
13.3.82 cusolverDnDsygvd_buffersize . . . 518
13.3.83 cusolverDnChegvd_buffersize . . . 518
13.3.84 cusolverDnZhegvd_buffersize . . . 518
13.3.85 cusolverDnSsygvd . . . 519
13.3.86 cusolverDnDsygvd . . . 519
13.3.87 cusolverDnChegvd . . . 519
13.3.88 cusolverDnZhegvd . . . 520
13.3.89 cusolverDnSsygvdx_buffersize . . . 520
13.3.90 cusolverDnDsygvdx_buffersize . . . 520
13.3.91 cusolverDnChegvdx_buffersize . . . 521
13.3.92 cusolverDnZhegvdx_buffersize . . . 521
13.3.93 cusolverDnSsygvdx . . . 521
13.3.94 cusolverDnDsygvdx . . . 522
13.3.95 cusolverDnChegvdx . . . 522
13.3.96 cusolverDnZhegvdx . . . 522
13.4 cusolverDn 64-bit API . . . 523
13.4.1 cusolverDnXpotrf_buffersize . . . 523
13.4.2 cusolverDnXpotrf . . . 523
13.4.3 cusolverDnXpotrs . . . 524
13.4.4 cusolverDnXgeqrf_buffersize . . . 524
13.4.5 cusolverDnXgeqrf . . . 525
13.4.6 cusolverDnXgetrf_buffersize . . . 525
13.4.7 cusolverDnXgetrf . . . 526
13.4.8 cusolverDnXgetrs . . . 526
13.4.9 cusolverDnXsyevd_buffersize . . . 527
13.4.10 cusolverDnXsyevd . . . 527
13.4.11 cusolverDnXsyevdx_buffersize . . . 528
13.4.12 cusolverDnXsyevdx . . . 528
13.4.13 cusolverDnXsytrs_bufferSize . . . 529
13.4.14 cusolverDnXsytrs . . . 529
13.4.15 cusolverDnXtrtri_bufferSize . . . 529
13.4.16 cusolverDnXtrtri . . . 530
13.4.17 cusolverDnXgesvd_buffersize . . . 530
13.4.18 cusolverDnXgesvd . . . 531
13.4.19 cusolverDnXgesvdp_buffersize . . . 531
13.4.20 cusolverDnXgesvdp . . . 532

13.4.21 cusolverDnXgesvdr_buffersize . . . 532
13.4.22 cusolverDnXgesvdr . . . 533
13.5 cusolverMp API . . . 533
13.5.1 cusolverMpCreate . . . 535
13.5.2 cusolverMpDestroy . . . 535
13.5.3 cusolverMpGetStream . . . 535
13.5.4 cusolverMpSetStream . . . 535
13.5.5 cusolverMpGetVersion . . . 535
13.5.6 cusolverMpCreateDeviceGrid . . . 536
13.5.7 cusolverMpDestroyGrid . . . 536
13.5.8 cusolverMpCreateMatrixDesc . . . 536
13.5.9 cusolverMpDestroyMatrixDesc . . . 536
13.5.10 cusolverMpNumROC . . . 537
13.5.11 cusolverMpBufferRegister . . . 537
13.5.12 cusolverMpBufferDeregister . . . 537
13.5.13 cusolverMpMalloc . . . 537
13.5.14 cusolverMpFree . . . 538
13.5.15 cusolverMpMatrixScatterH2D . . . 538
13.5.16 cusolverMpMatrixGatherD2H . . . 538
13.5.17 cusolverMpGetrf_buffersize . . . 539
13.5.18 cusolverMpGetrf . . . 539
13.5.19 cusolverMpGetrs_buffersize . . . 540
13.5.20 cusolverMpGetrs . . . 540
13.5.21 cusolverMpPotrf_buffersize . . . 541
13.5.22 cusolverMpPotrf . . . 541
13.5.23 cusolverMpPotrs_buffersize . . . 541
13.5.24 cusolverMpPotrs . . . 542
13.5.25 cusolverMpOrmqr_buffersize . . . 542
13.5.26 cusolverMpOrmqr . . . 543
13.5.27 cusolverMpOrmtr_buffersize . . . 543
13.5.28 cusolverMpOrmtr . . . 544
13.5.29 cusolverMpGels_buffersize . . . 544
13.5.30 cusolverMpGels . . . 545
13.5.31 cusolverMpStedc_buffersize . . . 545
13.5.32 cusolverMpStedc . . . 546
13.5.33 cusolverMpGeqrf_buffersize . . . 546
13.5.34 cusolverMpGeqrf . . . 547
13.5.35 cusolverMpOrgqr_buffersize . . . 547
13.5.36 cusolverMpOrgqr . . . 547
13.5.37 cusolverMpSytrd_buffersize . . . 548
13.5.38 cusolverMpSytrd . . . 548
13.5.39 cusolverMpSyevd_buffersize . . . 549
13.5.40 cusolverMpSyevd . . . 549
13.5.41 cusolverMpSygst_buffersize . . . 550
13.5.42 cusolverMpSygst . . . 550
13.5.43 cusolverMpSygvd_buffersize . . . 551
13.5.44 cusolverMpSygvd . . . 551
13.5.45 cusolverMpLaset . . . 552
13.5.46 cusolverMpNewtonSchulzDescriptorCreate . . . 552
13.5.47 cusolverMpNewtonSchulzDescriptorDestroy . . . 552
13.5.48 cusolverMpNewtonSchulzDescriptorSetAttribute . . . 552
13.5.49 cusolverMpNewtonSchulzDescriptorGetAttribute . . . 553
13.5.50 cusolverMpNewtonSchulz_buffersize . . . 553
13.5.51 cusolverMpNewtonSchulz . . . 553

13.5.52 cusolverMpPolarDescriptorCreate . . . 554
13.5.53 cusolverMpPolarDescriptorDestroy . . . 554
13.5.54 cusolverMpPolarDescriptorSetAttribute . . . 554
13.5.55 cusolverMpPolarDescriptorGetAttribute . . . 555
13.5.56 cusolverMpPolar_buffersize . . . 555
13.5.57 cusolverMpPolar . . . 555
13.5.58 cusolverMpGesvdDescriptorCreate . . . 556
13.5.59 cusolverMpGesvdDescriptorDestroy . . . 556
13.5.60 cusolverMpGesvdDescriptorSetAttribute . . . 556
13.5.61 cusolverMpGesvdDescriptorGetAttribute . . . 557
13.5.62 cusolverMpGesvd_buffersize . . . 557
13.5.63 cusolverMpGesvd . . . 558
13.5.64 cusolverMpSetMathMode . . . 558
13.5.65 cusolverMpGetMathMode . . . 559
13.5.66 cusolverMpSetEmulationStrategy . . . 559
13.5.67 cusolverMpGetEmulationStrategy . . . 559
13.5.68 cusolverMpLoggerSetFile . . . 559
13.5.69 cusolverMpLoggerOpenFile . . . 559
13.5.70 cusolverMpLoggerSetLevel . . . 560
13.5.71 cusolverMpLoggerSetMask . . . 560
13.5.72 cusolverMpLoggerForceDisable . . . 560
13.5.73 cusolverMpLoggerSetCallback . . . 560
13.6 NVLAMATH Runtime Library . . . 561
13.6.1 NVLAMATH Automatic Drop-In Acceleration . . . 561
13.6.2 NVLAMATH Usage From CUDA Fortran . . . 562
13.6.3 NVLAMATH Usage From OpenACC . . . 563
13.6.4 NVLAMATH Usage From OpenMP . . . 564
13.6.5 NVLAMATH List of Current Subroutines . . . 565
13.6.6 NVLAMATH Argument Checks and CPU Fallback . . . 565

14 Tensor Primitives Runtime Library APIs **567**
14.1 CUTENSOR Definitions and Helper Functions . . . 568
14.1.1 cutensorCreate . . . 574
14.1.2 cutensorDestroy . . . 574
14.1.3 cutensorCreateTensorDescriptor . . . 574
14.1.4 cutensorDestroyTensorDescriptor . . . 574
14.1.5 cutensorOperationDescriptorGetAttribute . . . 575
14.1.6 cutensorOperationDescriptorSetAttribute . . . 575
14.1.7 cutensorDestroyOperationDescriptor . . . 575
14.1.8 cutensorCreatePlanPreference . . . 575
14.1.9 cutensorDestroyPlanPreference . . . 576
14.1.10 cutensorPlanPreferenceSetAttribute . . . 576
14.1.11 cutensorPlanPreferenceGetAttribute . . . 576
14.1.12 cutensorEstimateWorkspaceSize . . . 576
14.1.13 cutensorCreatePlan . . . 577
14.1.14 cutensorDestroyPlan . . . 577
14.1.15 cutensorPlanGetAttribute . . . 577
14.1.16 cutensorGetErrorString . . . 577
14.1.17 cutensorGetVersion . . . 578
14.1.18 cutensorGetCudartVersion . . . 578
14.2 CUTENSOR Plan and Kernel Cache Operations . . . 578
14.2.1 cutensorHandleResizePlanCache . . . 578
14.2.2 cutensorHandleWritePlanCacheToFile . . . 579
14.2.3 cutensorHandleReadPlanCacheFromFile . . . 579

14.2.4 cutensorWriteKernelCacheToFile . . . 579
14.2.5 cutensorReadKernelCacheFromFile . . . 579
14.3 CUTENSOR Logging . . . 580
14.3.1 cutensorLoggerSetLevel . . . 580
14.3.2 cutensorLoggerSetMask . . . 580
14.3.3 cutensorLoggerOpenFile . . . 580
14.3.4 cutensorLoggerSetFile . . . 581
14.3.5 cutensorLoggerSetCallback . . . 581
14.3.6 cutensorLoggerForceDisable . . . 581
14.4 CUTENSOR Element-wise Operations . . . 581
14.4.1 cutensorCreatePermutation . . . 581
14.4.2 cutensorPermute . . . 582
14.4.3 cutensorCreateElementwiseBinary . . . 582
14.4.4 cutensorElementwiseBinaryExecute . . . 583
14.4.5 cutensorCreateElementwiseTrinary . . . 583
14.4.6 cutensorElementwiseTrinaryExecute . . . 583
14.5 CUTENSOR Reduction Operations . . . 584
14.5.1 cutensorCreateReduction . . . 584
14.5.2 cutensorReduce . . . 584
14.6 CUTENSOR Contraction Operations . . . 585
14.6.1 cutensorCreateContraction . . . 585
14.6.2 cutensorContract . . . 585
14.6.3 cutensorCreateContractionTrinary . . . 586
14.6.4 cutensorContractTrinary . . . 586
14.6.5 cutensorCreateBlockSparseTensorDescriptor . . . 586
14.6.6 cutensorDestroyBlockSparseTensorDescriptor . . . 587
14.6.7 cutensorCreateBlockSparseContraction . . . 587
14.6.8 cutensorBlockSparseContract . . . 588
14.7 CUTENSOR Fortran Extensions . . . 588
14.7.1 Fortran Reshape . . . 589
14.7.2 Fortran Transpose . . . 589
14.7.3 Fortran Spread . . . 589
14.7.4 Fortran Element-wise Expressions . . . 590
14.7.5 Fortran Matmul Operations . . . 590
14.7.6 Fortran Dot_Product Operations . . . 591
14.7.7 Supported Element-wise Functions . . . 592

15 NVIDIA Collective Communications Library (NCCL) APIs 593
15.1 NCCL Definitions and Helper Functions . . . 593
15.1.1 ncclGetLastError . . . 599
15.1.2 ncclGetErrorString . . . 599
15.1.3 ncclGetVersion . . . 599
15.1.4 ncclGetUniqueId . . . 599
15.1.5 ncclCommInitRank . . . 599
15.1.6 ncclCommInitAll . . . 600
15.1.7 ncclCommInitRankConfig . . . 600
15.1.8 ncclCommInitRankScalable . . . 600
15.1.9 ncclCommFinalize . . . 600
15.1.10 ncclCommDestroy . . . 601
15.1.11 ncclCommSplit . . . 601
15.1.12 ncclCommShrink . . . 601
15.1.13 ncclCommGetUniqueId . . . 601
15.1.14 ncclCommGrow . . . 602
15.1.15 ncclCommRevoke . . . 602

15.1.16 ncclCommAbort . . . 602
15.1.17 ncclCommRegister . . . 602
15.1.18 ncclCommDeregister . . . 603
15.1.19 ncclCommWindowRegister . . . 603
15.1.20 ncclCommWindowDeregister . . . 603
15.1.21 ncclWinGetUserPtr . . . 603
15.1.22 ncclCommGetAsyncError . . . 604
15.1.23 ncclCommCount . . . 604
15.1.24 ncclCommCuDevice . . . 604
15.1.25 ncclCommUserRank . . . 604
15.1.26 ncclCommQueryProperties . . . 604
15.1.27 ncclMemAlloc . . . 605
15.1.28 ncclMemFree . . . 605
15.1.29 ncclCommSuspend . . . 605
15.1.30 ncclCommResume . . . 605
15.1.31 ncclCommMemStats . . . 605
15.2 NCCL Collective Communication Functions . . . 606
15.2.1 ncclAllReduce . . . 606
15.2.2 ncclBroadcast . . . 606
15.2.3 ncclReduce . . . 607
15.2.4 ncclAllGather . . . 607
15.2.5 ncclReduceScatter . . . 607
15.2.6 ncclAlltoAll . . . 608
15.2.7 ncclGather . . . 608
15.2.8 ncclScatter . . . 609
15.2.9 ncclAllReduceConfig . . . 609
15.2.10 ncclBroadcastConfig . . . 609
15.2.11 ncclReduceConfig . . . 610
15.2.12 ncclAllGatherConfig . . . 610
15.2.13 ncclReduceScatterConfig . . . 611
15.2.14 ncclAlltoAllConfig . . . 611
15.2.15 ncclGatherConfig . . . 611
15.2.16 ncclScatterConfig . . . 612
15.3 NCCL Group Calls . . . 612
15.3.1 ncclGroupStart . . . 612
15.3.2 ncclGroupEnd . . . 613
15.3.3 ncclGroupSimulateEnd . . . 613
15.4 NCCL Point To Point Communication Functions . . . 613
15.4.1 Two-Sided Point-to-Point Operations . . . 613
15.4.1.1 ncclSend . . . 613
15.4.1.2 ncclRecv . . . 614
15.4.2 One-Sided Point-to-Point Operations (RMA) . . . 614
15.4.2.1 ncclPutSignal . . . 614
15.4.2.2 ncclSignal . . . 615
15.4.2.3 ncclWaitSignal . . . 615
15.5 User Defined Reduction Operators . . . 615
15.5.1 ncclRedOpCreatePreMulSum . . . 615
15.5.2 ncclRedOpDestroy . . . 616
15.6 NCCL Runtime Parameter Functions . . . 616
15.6.1 ncclParamBind . . . 616
15.6.2 Typed Parameter Getters . . . 616
15.6.3 ncclParamGetStr . . . 617
15.6.4 ncclParamGet . . . 617
15.6.5 ncclParamGetParameter . . . 617

15.6.6 ncclParamGetAllParameterKeys . . . 617
15.6.7 ncclParamDumpAll . . . 618

16 NVSHMEM Communication Library APIs 619
16.1 NVSHMEM Definitions, Setup, Exit, and Query Functions . . . 619
16.1.1 nvshmem_init . . . 621
16.1.2 nvshmemx_init_attr . . . 621
16.1.3 nvshmem_my_pe . . . 622
16.1.4 nvshmem_n_pes . . . 622
16.1.5 nvshmem_team_my_pe . . . 622
16.1.6 nvshmem_team_n_pes . . . 622
16.1.7 nvshmem_team_get_config . . . 622
16.1.8 nvshmem_team_translate_pe . . . 623
16.1.9 nvshmem_team_split_strided . . . 623
16.1.10 nvshmem_team_split_2d . . . 623
16.1.11 nvshmem_team_destroy . . . 623
16.1.12 nvshmem_info_get_version . . . 624
16.1.13 nvshmem_info_get_name . . . 624
16.1.14 nvshmem_finalize . . . 624
16.1.15 nvshmem_ptr . . . 624
16.2 NVSHMEM Memory Management Functions . . . 625
16.2.1 nvshmem_malloc . . . 625
16.2.2 nvshmem_free . . . 626
16.2.3 nvshmem_align . . . 626
16.2.4 nvshmem_calloc . . . 626
16.2.5 nvshmemx_buffer_register . . . 626
16.2.6 nvshmemx_buffer_unregister . . . 626
16.2.7 nvshmemx_buffer_unregister_all . . . 627
16.3 NVSHMEM Remote Memory Access Functions . . . 627
16.3.1 nvshmem_put . . . 627
16.3.2 nvshmem_p . . . 630
16.3.3 nvshmem_iput . . . 632
16.3.4 nvshmem_put_nbi . . . 635
16.3.5 nvshmemx_put_block . . . 638
16.3.6 nvshmemx_put_warp . . . 640
16.3.7 nvshmemx_put_nbi_block . . . 641
16.3.8 nvshmemx_put_nbi_warp . . . 642
16.3.9 nvshmemx_iput_block . . . 643
16.3.10 nvshmemx_iput_warp . . . 643
16.3.11 nvshmem_get . . . 644
16.3.12 nvshmem_g . . . 647
16.3.13 nvshmem_iget . . . 648
16.3.14 nvshmem_get_nbi . . . 651
16.3.15 nvshmemx_get_block . . . 654
16.3.16 nvshmemx_get_warp . . . 656
16.3.17 nvshmemx_get_nbi_block . . . 657
16.3.18 nvshmemx_get_nbi_warp . . . 658
16.3.19 nvshmemx_iget_block . . . 659
16.3.20 nvshmemx_iget_warp . . . 659
16.3.21 nvshmemx_put_signal_block . . . 660
16.3.22 nvshmemx_put_signal_warp . . . 660
16.3.23 nvshmemx_put_signal_on_stream . . . 661
16.3.24 nvshmemx_put_signal_nbi_block . . . 662
16.3.25 nvshmemx_put_signal_nbi_warp . . . 663

16.3.26 nvshmemx_put_signal_nbi_on_stream . . . 663
16.4 NVSHMEM Collective Communication Functions . . . 664
16.4.1 nvshmem_barrier, nvshmem_barrier_all . . . 664
16.4.2 nvshmem_sync, nvshmem_sync_all . . . 665
16.4.3 nvshmem_alltoall . . . 665
16.4.4 nvshmem_broadcast . . . 666
16.4.5 nvshmem_collect . . . 669
16.4.6 NVSHMEM Reductions . . . 672
16.4.6.1 nvshmem_and_reduce . . . 672
16.4.6.2 nvshmem_or_reduce . . . 674
16.4.6.3 nvshmem_xor_reduce . . . 676
16.4.6.4 nvshmem_max_reduce . . . 677
16.4.6.5 nvshmem_min_reduce . . . 680
16.4.6.6 nvshmem_sum_reduce . . . 683
16.4.6.7 nvshmem_prod_reduce . . . 685
16.4.7 NVSHMEM Reduce-Scatter . . . 688
16.4.7.1 nvshmem_and_reducescatter . . . 688
16.4.7.2 nvshmem_or_reducescatter . . . 689
16.4.7.3 nvshmem_xor_reducescatter . . . 690
16.4.7.4 nvshmem_max_reducescatter . . . 691
16.4.7.5 nvshmem_min_reducescatter . . . 692
16.4.7.6 nvshmem_sum_reducescatter . . . 693
16.4.7.7 nvshmem_prod_reducescatter . . . 694
16.4.7.8 nvshmemx_double2_maxloc_reduce_block . . . 695
16.5 NVSHMEM Atomic Memory Operations . . . 695
16.5.1 nvshmem_atomic_inc . . . 695
16.5.2 nvshmem_atomic_fetch_inc . . . 695
16.5.3 nvshmem_atomic_fetch . . . 696
16.5.4 nvshmem_atomic_set . . . 696
16.5.5 nvshmem_atomic_add . . . 696
16.5.6 nvshmem_atomic_fetch_add . . . 696
16.5.7 nvshmem_atomic_swap . . . 697
16.5.8 nvshmem_atomic_compare_swap . . . 697
16.5.9 nvshmem_atomic_and . . . 697
16.5.10 nvshmem_atomic_fetch_and . . . 697
16.5.11 nvshmem_atomic_or . . . 698
16.5.12 nvshmem_atomic_fetch_or . . . 698
16.5.13 nvshmem_atomic_xor . . . 698
16.5.14 nvshmem_atomic_fetch_xor . . . 698
16.5.15 nvshmemx_atomic_add . . . 699
16.5.16 nvshmemx_atomic_fetch_add . . . 699
16.6 NVSHMEM Point to Point Synchronization Functions . . . 699
16.6.1 nvshmem_wait_until . . . 699
16.6.2 nvshmem_wait_until_all . . . 700
16.6.3 nvshmem_wait_until_any . . . 700
16.6.4 nvshmem_wait_until_some . . . 701
16.6.5 nvshmem_wait_until_all_vector . . . 701
16.6.6 nvshmem_wait_until_any_vector . . . 702
16.6.7 nvshmem_wait_until_some_vector . . . 702
16.6.8 nvshmem_test . . . 703
16.6.9 nvshmem_test_all . . . 703
16.6.10 nvshmem_test_any . . . 703
16.6.11 nvshmem_test_some . . . 704
16.6.12 nvshmem_test_all_vector . . . 704

16.6.13 nvshmem_test_any_vector . . . 705
16.6.14 nvshmem_test_some_vector . . . 705
16.6.15 nvshmemx_signal_counted_reset . . . 706
16.7 NVSHMEM Memory Ordering Functions . . . 706
16.7.1 nvshmem_fence . . . 706
16.7.2 nvshmem_quiet . . . 706
16.7.3 nvshmemx_flush . . . 706
16.8 NVSHMEM Region Functions . . . 707
16.8.1 nvshmemx_region_start, nvshmemx_region_stop . . . 707
16.8.2 nvshmemx_region_start_block, nvshmemx_region_stop_block . . . 707
16.8.3 nvshmemx_region_is_active . . . 708

17 NVTX Profiling Library APIs **709**
17.1 NVTX Basic Tooling APIs . . . 709
17.1.1 nvtxStartRange . . . 710
17.1.2 nvtxEndRange . . . 710
17.2 NVTX Advanced Tooling APIs . . . 710
17.2.1 NVTX Definitions and Derived Types . . . 710
17.2.2 nvtxInitialize . . . 712
17.2.3 nvtxDomainCreate . . . 713
17.2.4 nvtxDomainDestroy . . . 713
17.2.5 nvtxDomainRegisterString . . . 713
17.2.6 nvtxDomainNameCategory . . . 714
17.2.7 nvtxNameCategory . . . 714
17.2.8 nvtxNameOsThread . . . 714
17.2.9 nvtxDomainResourceCreate . . . 714
17.2.10 nvtxDomainResourceDestroy . . . 715
17.2.11 nvtxDomainMarkEx . . . 715
17.2.12 nvtxMarkEx . . . 715
17.2.13 nvtxMark . . . 715
17.2.14 nvtxDomainRangeStartEx . . . 715
17.2.15 nvtxRangeStartEx . . . 716
17.2.16 nvtxRangeStart . . . 716
17.2.17 nvtxDomainRangeEnd . . . 716
17.2.18 nvtxRangeEnd . . . 716
17.2.19 nvtxDomainRangePushEx . . . 717
17.2.20 nvtxRangePushEx . . . 717
17.2.21 nvtxRangePush . . . 717
17.2.22 nvtxDomainRangePop . . . 717
17.2.23 nvtxRangePop . . . 718
17.3 NVTX Automated Instrumentation . . . 718

18 NVIDIA Communication Abstraction Library (CAL) APIs **719**
18.1 CAL API . . . 719
18.1.1 cal_comm_create_mpi . . . 720
18.1.2 cal_comm_destroy . . . 720
18.1.3 cal_stream_sync . . . 720
18.1.4 cal_comm_barrier . . . 721
18.1.5 cal_comm_get_rank . . . 721
18.1.6 cal_comm_get_size . . . 721

19 Examples **723**
19.1 Using cuBLAS from OpenACC Host Code . . . 723
19.2 Using cuBLAS from CUDA Fortran Host Code . . . 725

19.3 Using cuFFT from OpenACC Host Code 728
19.4 Using cuFFT from CUDA Fortran Host Code 729
19.5 Using cufftXt from CUDA Fortran Host Code 730
19.6 Using cuFFTMp from either OpenACC or CUDA Fortran 736
19.7 Using cuRAND from OpenACC Host Code 744
19.8 Using cuRAND from OpenACC Device Code 747
19.9 Using cuRAND from CUDA Fortran Host Code 748
19.10 Using cuRAND from CUDA Fortran Device Code 750
19.11 Using cuSPARSE from OpenACC Host Code 752
19.12 Using cuSPARSE from CUDA Fortran Host Code 754
19.13 Using cuTENSOR from CUDA Fortran Host Code 756
19.14 Using cuTENSOREX from CUDA Fortran Host Code 758
19.15 Using cuTENSOR from OpenACC Host Code 760

NVIDIA Fortran CUDA Interfaces

Preface

This document describes the NVIDIA Fortran interfaces to cuBLAS, cuFFT, cuRAND, cuSPARSE, and other CUDA Libraries used in scientific and engineering applications built upon the CUDA computing architecture.

Intended Audience

This guide is intended for application programmers, scientists and engineers proficient in programming with the Fortran language. This guide assumes some familiarity with either CUDA Fortran or OpenACC.

Organization

The organization of this document is as follows:

Introduction
contains a general introduction to Fortran interfaces, OpenACC, CUDA Fortran, and CUDA Library functions

BLAS Runtime Library
describes the Fortran interfaces to the various cuBLAS libraries

FFT Runtime Library APIs
describes the module types, definitions and Fortran interfaces to the cuFFT library

Random Number Runtime APIs
describes the Fortran interfaces to the host and device cuRAND libraries

Sparse Matrix Runtime APIs
describes the module types, definitions and Fortran interfaces to the cuSPARSE Library

Matrix Solver Runtime APIs
describes the module types, definitions and Fortran interfaces to the cuSOLVER Library

Tensor Primitives Runtime APIs
describes the module types, definitions and Fortran interfaces to the cuTENSOR Library

NVIDIA Collective Communications Library APIs
describes the module types, definitions and Fortran interfaces to the NCCL Library

NVSHMEM Communication Library APIs
describes the module types, definitions and Fortran interfaces to the NVSHMEM Library

NVTX Profiling Library APIs
describes the module types, definitions and Fortran interfaces to the NVTX API and Library

Examples
provides sample code and an explanation of each of the simple examples.

Conventions

This guide uses the following conventions:

italic
is used for emphasis.

`Constant Width`
is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
is used for commands.

[item1]
in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ item2 | item 3 }
braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...
ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

`FORTRAN`
Fortran language statements are shown in the text of this guide using a reduced fixed point size.

`C++ and C`
C++ and C language statements are shown in the text of this guide using a reduced fixed point size.

Terminology

If there are terms in this guide with which you are unfamiliar, see the NVIDIA HPC glossary.

Related Publications

The following documents contain additional information related to OpenACC and CUDA Fortran programming, CUDA, and the CUDA Libraries.

- ISO/IEC 1539-1:1997, Information Technology – Programming Languages – FORTRAN, Geneva, 1997 (Fortran 95).
- NVIDIA CUDA Programming Guides, NVIDIA. Available online at docs.nvidia.com/cuda.
- NVIDIA HPC Compiler User's Guide, Release 2024. Available online at docs.nvidia.com/hpc-sdk.

This document provides a reference for calling CUDA Library functions from NVIDIA Fortran. It can be used from Fortran code using the OpenACC or OpenMP programming models, or from NVIDIA CUDA Fortran. Currently, the CUDA libraries which NVIDIA provides pre-built interface modules for, and which are documented here, are:

- cuBLAS, an implementation of the BLAS.
- cuFFT, a library of Fast Fourier Transform (FFT) routines.
- cuRAND, a library for random number generation.
- cuSPARSE, a library of linear algebra routines used with sparse matrices.

- cuSOLVER, a library of equation solvers used with dense or other matrices.
- cuTENSOR, a library for tensor primitive operations.
- NCCL, a collective communications libraries.
- NVSHMEM, a library implementation of OpenSHMEM on GPUs.
- NVTX, an API for annotating application events, code ranges, and resources.

The OpenACC Application Program Interface is a collection of compiler directives and runtime routines that allows the programmer to specify loops and regions of code for offloading from a host CPU to an attached accelerator, such as a GPU. The OpenACC API was designed and is maintained by an industry consortium. See the OpenACC website for more information about the OpenACC API.

OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify parallel execution from Fortran (and other languages). The OpenMP target offload capabilities are similar in many respects to OpenACC. The methods for passing device arrays to library functions from host code differ only in syntax compared to those used in OpenACC. For general information about using OpenMP and to obtain a copy of the OpenMP specification, refer to the OpenMP organization's website.

CUDA Fortran is a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture. CUDA Fortran includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran, and is an analog to NVIDIA's CUDA C compiler. Compared to the NVIDIA Accelerator and OpenACC directives-based model and compilers, CUDA Fortran is a lower-level explicit programming model with substantial runtime library components that give expert programmers direct control of all aspects of GPGPU programming.

This document does not contain explanations or purposes of the library functions, nor does it contain details of the approach used in the CUDA implementation to target GPUs. For that information, please see the appropriate library document that comes with the NVIDIA CUDA Toolkit. This document does provide the Fortran module contents: derived types, enumerations, and interfaces, to make use of the libraries from Fortran rather than from C or C++.

Many of the examples used in this document are provided in the HPC compiler and tools distribution, along with Makefiles, and are stored in the yearly directory, such as 2020/examples/CUDA-Libraries.

Chapter 1. Fortran Interfaces and Wrappers

Almost all of the function interfaces shown in this document make use of features from the Fortran 2003 iso_c_binding intrinsic module. This module provides a standard way for dealing with isues such as inter-language data types, capitalization, adding underscores to symbol names, or passing arguments by value.

Often, the iso_c_binding module enables Fortran programs containing properly written interfaces to call directly into the C library functions. In some cases, NVIDIA has written small wrappers around the C library function, to make the Fortran call site more “Fortran-like”, hiding some issues exposed in the C interfaces like handle management, host vs. device pointer management, or character and complex data type issues.

In a small number of cases, the C Library may contain multiple entry points to handle different data types, perhaps an int in one function and a size_t in another, otherwise the functions are identical. In these cases, NVIDIA may provide just one generic Fortran interface, and will call the appropriate C function under the hood.

Chapter 2. Using CUDA Libraries from OpenACC Host Code

All of the libraries covered in this document contain functions which are callable from OpenACC host code. Most functions take some arguments which are expected to be device pointers (the address of a variable in device global memory). There are several ways to do that in OpenACC.

If the call is lexically nested within an OpenACC data directive, the NVIDIA Fortran compiler, in the presence of an explicit interface such as those provided by the NVIDIA library modules, will default to passing the device pointer when required.

```
subroutine hostcall(a, b, n)
use cublas
real a(n), b(n)
!$acc data copy(a, b)
call cublasSswap(n, a, 1, b, 1)
!$acc end data

return
end
```

A Fortran interface is made explicit when you use the module that contains it, as in the line use cublas in the example above. If you look ahead to the actual interface for cublasSswap, you will see that the arrays a and b are declared with the CUDA Fortran device attribute, so they take only device addresses as arguments.

It is more acceptable and general when using OpenACC to pass device pointers to subprograms by using the host_data clause as most implementations don't have a way to mark arguments as device pointers. The host_data construct with the use_device clause makes the device addresses available in host code for passing to the subprogram.

```
use cufft
use openacc
. . .
!$acc data copyin(a), copyout(b,c)
ierr = cufftPlan2D(iplan1,m,n,CUFFT_C2C)
ierr = ierr + cufftSetStream(iplan1,acc_get_cuda_stream(acc_async_sync))
!$acc host_data use_device(a,b,c)
ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
!$acc end host_data

! scale c
!$acc kernels
```

(continues on next page)

(continued from previous page)

```
c = c / (m*n)
!$acc end kernels
!$acc end data
```

This code snippet also shows an example of sharing the stream that OpenACC and the cuFFT library use. Every library in this document has a function for setting the CUDA stream which the library runs on. Usually, when using OpenACC, you want the OpenACC kernels to run on the same stream as the library functions. In the case above, this guarantees that the kernel `c = c / (m*n)` does not start until the FFT operations complete. The function acc_get_cuda_stream and the definition for acc_async_sync are in the openacc module.

Chapter 3. Using CUDA Libraries from OpenACC Device Code

Two libraries are currently available from within OpenACC compute regions. Certain functions in both the openacc_curand module and the nvshmem module are marked `acc routine seq`.

The cuRAND device library is all contained within CUDA header files. In device code, it is designed to return one or a small number of random numbers per thread. The thread's random generators run independently of each other, and it is usually advised for performance reasons to give each thread a different seed, rather than a different offset.

```
program t
use openacc_curand
integer, parameter :: n = 500
real a(n,n,4)
type(curandStateXORWOW) :: h
integer(8) :: seed, seq, offset
a = 0.0
!$acc parallel num_gangs(n) vector_length(n) copy(a)
!$acc loop gang
do j = 1, n
!$acc loop vector private(h)
  do i = 1, n
    seed = 12345_8 + j*n*n + i*2
    seq = 0_8
    offset = 0_8
    call curand_init(seed, seq, offset, h)
!$acc loop seq
    do k = 1, 4
      a(i,j,k) = curand_uniform(h)
    end do
  end do
end do
!$acc end parallel
print *,maxval(a),minval(a),sum(a)/(n*n*4)
end
```

When using the openacc_curand module, since all the code is contained in CUDA header files, you do not need any additional libraries on the link line.

Chapter 4. Using CUDA Libraries from CUDA Fortran Host Code

The predominant usage model for the library functions listed in this document is to call them from CUDA Host code. CUDA Fortran allows some special capabilities in that the compiler is able to recognize the device and managed attribute in resolving generic interfaces. Device actual arguments can only match the interface's device dummy arguments; managed actual arguments, by precedence, match managed dummy arguments first, then device dummies, then host.

```
program testisamax  ! link with -cudalib=cublas -lblas
use cublas
real*4              x(1000)
real*4, device  :: xd(1000)
real*4, managed :: xm(1000)

call random_number(x)

! Call host BLAS
j = isamax(1000,x,1)

xd = x
! Call cuBLAS
k = isamax(1000,xd,1)
print *,j.eq.k

xm = x
! Also calls cuBLAS
k = isamax(1000,xm,1)
print *,j.eq.k
end
```

Using the `cudafor` module, the full set of CUDA functionality is available to programmers for managing CUDA events, streams, synchronization, and asynchronous behaviors. CUDA Fortran can be used in OpenMP programs, and the CUDA Libraries in this document are thread safe with respect to host CPU threads. Further examples are included in chapter Examples.

Chapter 5. Using CUDA Libraries from CUDA Fortran Device Code

The cuRAND and NVSHMEM libraries have functions callable from CUDA Fortran device code, and their interfaces are accessed via the curand_device and nvshmem modules, respectively. The module interfaces are very similar to the modules used in OpenACC device code, but for CUDA Fortran, each subroutine and function is declared attributes([host,]device), and the subroutines and functions do not need to be marked as `acc routine seq`.

```
module mrand
    use curand_device
    integer, parameter :: n = 500
    contains
    attributes(global) subroutine randsub(a)
    real, device :: a(n,n,4)
    type(curandStateXORWOW) :: h
    integer(8) :: seed, seq, offset
    j = blockIdx%x; i = threadIdx%x
    seed = 12345_8 + j*n*n + i*2
    seq = 0_8
    offset = 0_8
    call curand_init(seed, seq, offset, h)
    do k = 1, 4
        a(i,j,k) = curand_uniform(h)
    end do
    end subroutine
end module

program t   ! nvfortran t.cuf
use mrand
use cudafor ! recognize maxval, minval, sum w/managed
real, managed :: a(n,n,4)
a = 0.0
call randsub<<<n,n>>>(a)
print *,maxval(a),minval(a),sum(a)/(n*n*4)
end program
```

Chapter 6. Pointer Modes in cuBLAS and cuSPARSE

Because the NVIDIA Fortran compiler can distinguish between host and device arguments, the NVIDIA modules for interfacing to cuBLAS and cuSPARSE handle pointer modes differently than CUDA C, which requires setting the mode explicitly for scalar arguments. Examples of scalar arguments which can reside either on the host or device are the alpha and beta scale factors to the *gemm functions.

Typically, when using the normal “non-_v2” interfaces in the cuBLAS and cuSPARSE modules, the run-time wrappers will implicitly add the setting and restoring of the library pointer modes behind the scenes. This adds some negligible but non-zero overhead to the calls.

To avoid the implicit getting and setting of the pointer mode with every invocation of a library function do the following:

- For the BLAS, use the `cublas_v2` module, and the v2 entry points, such as `cublasIsamax_v2`. It is the programmer’s responsibility to properly set the pointer mode when needed. Examples of scalar arguments which do require setting the pointer mode are the alpha and beta scale factors passed to the *gemm routines, and the scalar results returned from the v2 versions of the *amax(), *amin(), *asum(), *rotg(), *rotmg(), *nrm2(), and *dot() functions. In the v2 interfaces shown in the chapter 2, these scalar arguments will have the comment `! device or host variable`. Examples of scalar arguments which do not require setting the pointer mode are increments, extents, and lengths such as incx, incy, n, lda, ldb, and ldc.
- For the cuSPARSE library, each function listed in chapter 5 which contains scalar arguments with the comment `! device or host variable` has a corresponding v2 interface, though it is not documented here. For instance, in addition to the interface named `cusparseSaxpyi`, there is another interface named `cusparseSaxpyi_v2` with the exact same argument list which calls into the cuSPARSE library directly and will not implicitly get or set the library pointer mode.

The CUDA default pointer mode is that the scalar arguments reside on the host. The NVIDIA runtime does not change that setting.

Chapter 7. Writing Your Own CUDA Interfaces

Despite the large number of interfaces included in the modules described in this document, users will have the need from time-to-time to write their own interfaces to new libraries or their own tuned CUDA, perhaps written in C/C++. There are some standard techniques to use, and some non-standard NVIDIA extensions which can make creating working interfaces easier.

```
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction ) &
        bind(C,name='cufftExecC2C')
        integer, value :: plan
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
end interface cufftExecC2C
```

This interface calls the C library function directly. You can deal with Fortran's capitalization issues by putting the properly capitalized C function in the `bind(C)` attribute. If the C function expects input arguments passed by value, you can add the `value` attribute to the dummy declaration as well. A nice feature of Fortran is that the interface can change, but the code at the call site may not have to. The compiler changes the details of the call to fit the interface.

Now suppose a user of this interface would like to call this function with REAL data (F77 code is notorious for mixing REAL and COMPLEX declarations). There are two ways to do this:

```
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction ) &
        bind(C,name='cufftExecC2C')
        integer, value :: plan
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
    integer function cufftExecR2R( plan, idata, odata, direction ) &
        bind(C,name='cufftExecC2C')
        integer, value :: plan
        real, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecR2R
end interface cufftExecC2C
```

Here the C name hasn't changed. The compiler will now accept actual arguments corresponding to idata and odata that are declared REAL. A generic interface is created named `cufftExecC2C`. If you

have problems debugging your generic interface, as a debugging aid you can try calling the specific name, `cufftExecR2R` in this case, to help diagnose the problem.

A commonly used extension which is supported by NVIDIA is `ignore_tkr`. A programmer can use it in an interface to instruct the compiler to ignore any combination of the type, kind, and rank during the interface matching process. The previous example using `ignore_tkr` looks like this:

```
! cufftExecC2C
interface cufftExecC2C
    integer function cufftExecC2C( plan, idata, odata, direction ) &
        bind(C,name='cufftExecC2C')
        integer, value :: plan
        !dir$ ignore_tkr(tr) idata, (tr) odata
        complex, device, dimension(*) :: idata, odata
        integer, value :: direction
    end function cufftExecC2C
end interface cufftExecC2C
```

Now the compiler will ignore both the type and rank (F77 could also be sloppy in its handling of array dimensions) of idata and odata when matching the call site to the interface. An unfortunate side-effect is that the interface will now allow integer, logical, and character data for idata and odata. It is up to the implementor to determine if that is acceptable.

A final aid, specific to NVIDIA, worth mentioning here is `ignore_tkr (d)`, which ignores the device attribute of an actual argument during interface matching.

Of course, if you write a wrapper, a narrow strip of code between the Fortran call and your library function, you are not limited by the simple transormations that a compiler can do, such as those listed here. As mentioned earlier, many of the interfaces provided in the cuBLAS and cuSPARSE modules use wrappers.

A common request is a way for Fortran programmers to take advantage of the thrust library. Explaining thrust and C++ programming is outside of the scope of this document, but this simple example can show how to take advantage of the excellent sort capabilities in thrust:

```
// Filename: csort.cu
// nvcc -c -arch sm_35 csort.cu
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/sort.h>

extern "C" {

    //Sort for integer arrays
    void thrust_int_sort_wrapper( int *data, int N)
    {
    thrust::device_ptr <int> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for float arrays
    void thrust_float_sort_wrapper( float *data, int N)
    {
    thrust::device_ptr <float> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for double arrays
```

(continues on next page)

(continued from previous page)

```
    void thrust_double_sort_wrapper( double *data, int N)
    {
    thrust::device_ptr <double> dev_ptr(data);
    thrust::sort(dev_ptr, dev_ptr+N);
    }
}
```

Set up interface to the sort subroutine in Fortran and calls are simple:

```
program t
interface sort
   subroutine sort_int(array, n) &
        bind(C,name='thrust_int_sort_wrapper')
        integer(4), device, dimension(*) :: array
        integer(4), value :: n
    end subroutine
end interface
integer(4), parameter :: n = 100
integer(4), device :: a_d(n)
integer(4) :: a_h(n)
!$cuf kernel do
do i = 1, n
    a_d(i) = 1 + mod(47*i,n)
end do
call sort(a_d, n)
a_h = a_d
nres  = count(a_h .eq. (/(i,i=1,n)/))
if (nres.eq.n) then
    print *,"test PASSED"
else
    print *,"test FAILED"
endif
end
```

Chapter 8. NVIDIA Fortran Compiler Options

The NVIDIA Fortran compiler driver is called nvfortran. General information on the compiler options which can be passed to nvfortran can be obtained by typing nvfortran -help. To enable targeting NVIDIA GPUs using OpenACC, use nvfortran -acc=gpu. To enable targeting NVIDIA GPUs using CUDA Fortran, use nvfortran -cuda. CUDA Fortran is also supported by the NVIDIA Fortran compilers when the filename uses the .cuf extension. Uppercase file extensions, .F90 or .CUF, for example, may also be used, in which case the program is processed by the preprocessor before being compiled.

Other options which are pertinent to the examples in this document are:

- -cudalib[=cublas|cufft|cufftw|curand|cusolver|cusparse|cutensor|nvblas|nccl|nvshmem|nvlamath|nvtx]: this option adds the appropriate versions of the CUDA-optimized libraries to the link line. It handles static and dynamic linking, and platform (Linux, Windows) differences unobtrusively.
- -gpu=cc70: this option compiles for compute capability 7.0. Certain library functionality may require minimum compute capability of 6.0, 7.0, or higher.
- -gpu=cudaX.Y: this option compiles and links with a particular CUDA Toolkit version. Certain library functionality may require a newer (or older, for deprecated functions) CUDA runtime version.

Chapter 9. BLAS Runtime APIs

This section describes the Fortran interfaces to the CUDA BLAS libraries. There are currently four separate collections of function entry points which are commonly referred to as the cuBLAS:

- The original CUDA implementation of the BLAS routines, referred to as the legacy API, which are callable from the host and expect and operate on device data.
- The newer "v2" CUDA implementation of the BLAS routines, plus some extensions for batched operations. These are also callable from the host and operate on device data. In Fortran terms, these entry points have been changed from subroutines to functions which return status.
- The cuBLAS XT library which can target multiple GPUs using only host-resident data.
- The cuBLAS MP library which can target multiple GPUs using distributed device data, similar to the ScaLAPACK PBLAS functions. The cublasMp and cusolverMp libraries are built, in part, upon a communications library named CAL, which is documented in another section of this document.

NVIDIA currently ships with four Fortran modules which programmers can use to call into this cuBLAS functionality:

- cublas, which provides interfaces to into the main cublas library. Both the legacy and v2 names are supported. In this module, the cublas names (such as cublasSaxpy) use the legacy calling conventions. Interfaces to a host BLAS library (for instance libblas.a in the NVIDIA distribution) are also included in the cublas module. These interfaces are exposed by adding the line

  ```
  use cublas
  ```

  to your program unit.
- cublas_v2, which is similar to the cublas module in most ways except the cublas names (such as cublasSaxpy) use the v2 calling conventions. For instance, instead of a subroutine, cublasSaxpy is a function which takes a handle as the first argument and returns an integer containing the status of the call. These interfaces are exposed by adding the line

  ```
  use cublas_v2
  ```

  to your program unit.
- cublasxt, which interfaces directly to the cublasXT API. These interfaces are exposed by adding the line

  ```
  use cublasxt
  ```

  to your program unit.
- cublasmp, which provides interfaces into the cublasMp API. These interfaces are exposed by adding the line

```
use cublasMp
```

to your program unit.

The v2 routines are integer functions that return an error status code; they return a value of CUBLAS_STATUS_SUCCESS if the call was successful, or other cuBLAS status return value if there was an error.

Documented interfaces to the traditional BLAS names in the subsequent sections, which contain the comment `! device or host variable` should not be confused with the pointer mode issue from section 1.6. The traditional BLAS names are overloaded generic names in the `cublas` module. For instance, in this interface

```
subroutine scopy(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

The arrays x and y can either both be device arrays, in which case `cublasScopy` is called via the generic interface, or they can both be host arrays, in which case `scopy` from the host BLAS library is called. Using CUDA Fortran managed data as actual arguments to `scopy` poses an interesting case, and calling `cublasScopy` is chosen by default. If you wish to call the host library version of scopy with managed data, don't expose the generic scopy interface at the call site.

Unless a specific kind is provided, in the following interfaces the plain integer type implies integer(4) and the plain real type implies real(4).

9.1. CUBLAS Definitions and Helper Functions

This section contains definitions and data types used in the cuBLAS library and interfaces to the cuBLAS Helper Functions.

The cublas module contains the following derived type definitions:

```
TYPE cublasHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

The cuBLAS module contains the following enumerations:

```
enum, bind(c)
    enumerator :: CUBLAS_STATUS_SUCCESS         =0
    enumerator :: CUBLAS_STATUS_NOT_INITIALIZED =1
    enumerator :: CUBLAS_STATUS_ALLOC_FAILED    =3
    enumerator :: CUBLAS_STATUS_INVALID_VALUE   =7
    enumerator :: CUBLAS_STATUS_ARCH_MISMATCH   =8
    enumerator :: CUBLAS_STATUS_MAPPING_ERROR   =11
    enumerator :: CUBLAS_STATUS_EXECUTION_FAILED=13
    enumerator :: CUBLAS_STATUS_INTERNAL_ERROR  =14
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_FILL_MODE_LOWER=0
    enumerator :: CUBLAS_FILL_MODE_UPPER=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_DIAG_NON_UNIT=0
    enumerator :: CUBLAS_DIAG_UNIT=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_SIDE_LEFT =0
    enumerator :: CUBLAS_SIDE_RIGHT=1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_OP_N=0
    enumerator :: CUBLAS_OP_T=1
    enumerator :: CUBLAS_OP_C=2
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_POINTER_MODE_HOST   = 0
    enumerator :: CUBLAS_POINTER_MODE_DEVICE = 1
end enum
```

9.1.1. cublasCreate

This function initializes the CUBLAS library and creates a handle to an opaque structure holding the CUBLAS library context. It allocates hardware resources on the host and device and must be called prior to making any other CUBLAS library calls. The CUBLAS library context is tied to the current CUDA device. To use the library on multiple devices, one CUBLAS handle needs to be created for each device. Furthermore, for a given device, multiple CUBLAS handles with different configuration can be created. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences. For multi-threaded applications that use the same device from different threads, the recommended programming model is to create one CUBLAS handle per thread and use that CUBLAS handle for the entire life of the thread.

```
integer(4) function cublasCreate(handle)
  type(cublasHandle) :: handle
```

9.1.2. cublasDestroy

This function releases hardware resources used by the CUBLAS library. This function is usually the last call with a particular handle to the CUBLAS library. Because cublasCreate allocates some internal resources and the release of those resources by calling cublasDestroy will implicitly call cublasDeviceSynchronize, it is recommended to minimize the number of cublasCreate/cublasDestroy occurences.

```
integer(4) function cublasDestroy(handle)
  type(cublasHandle) :: handle
```

9.1.3. cublasGetVersion

This function returns the version number of the cuBLAS library.

```
integer(4) function cublasGetVersion(handle, version)
  type(cublasHandle) :: handle
  integer(4) :: version
```

9.1.4. cublasSetStream

This function sets the cuBLAS library stream, which will be used to execute all subsequent calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream. In particular, this routine can be used to change the stream between kernel launches and then to reset the cuBLAS library stream back to NULL.

```
integer(4) function cublasSetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

9.1.5. cublasGetStream

This function gets the cuBLAS library stream, which is being used to execute all calls to the cuBLAS library functions. If the cuBLAS library stream is not set, all kernels use the default NULL stream.

```
integer(4) function cublasGetStream(handle, stream)
  type(cublasHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

9.1.6. cublasGetStatusName

This function returns the cuBLAS status name associated with a given status value.

```
character(128) function cublasGetStatusName(ierr)
  integer(4) :: ierr
```

9.1.7. cublasGetStatusString

This function returns the cuBLAS status string associated with a given status value.

```
character(128) function cublasGetStatusString(ierr)
  integer(4) :: ierr
```

9.1.8. cublasGetPointerMode

This function obtains the pointer mode used by the cuBLAS library. In the `cublas` module, the pointer mode is set and reset on a call-by-call basis depending on the whether the device attribute is set on scalar actual arguments. See section 1.6 for a discussion of pointer modes.

```
integer(4) function cublasGetPointerMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.9. cublasSetPointerMode

This function sets the pointer mode used by the cuBLAS library. When using the `cublas` module, the pointer mode is set on a call-by-call basis depending on the whether the device attribute is set on scalar actual arguments. When using the `cublas_v2` module with v2 interfaces, it is the programmer's responsibility to make calls to `cublasSetPointerMode` so scalar arguments are handled correctly by the library. See section 1.6 for a discussion of pointer modes.

```
integer(4) function cublasSetPointerMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.10. cublasGetAtomicsMode

This function obtains the atomics mode used by the cuBLAS library.

```
integer(4) function cublasGetAtomicsMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.11. cublasSetAtomicsMode

This function sets the atomics mode used by the cuBLAS library. Some routines in the cuBLAS library have alternate implementations that use atomics to accumulate results. These alternate implementations may run faster but may also generate results which are not identical from one run to the other. The default is to not allow atomics in cuBLAS functions.

```
integer(4) function cublasSetAtomicsMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.12. cublasGetMathMode

This function obtains the math mode used by the cuBLAS library.

```
integer(4) function cublasGetMathMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.13. cublasSetMathMode

This function sets the math mode used by the cuBLAS library. Some routines in the cuBLAS library allow you to choose the compute precision used to generate results. These alternate approaches may run faster but may also generate different, less accurate results.

```
integer(4) function cublasSetMathMode(handle, mode)
  type(cublasHandle) :: handle
  integer(4) :: mode
```

9.1.14. cublasGetSmCountTarget

This function obtains the SM count target used by the cuBLAS library.

```
integer(4) function cublasGetSmCountTarget(handle, counttarget)
  type(cublasHandle) :: handle
  integer(4) :: counttarget
```

9.1.15. cublasSetSmCountTarget

This function sets the SM count target used by the cuBLAS library.

```
integer(4) function cublasSetSmCountTarget(handle, counttarget)
  type(cublasHandle) :: handle
  integer(4) :: counttarget
```

9.1.16. cublasGetHandle

This function gets the cuBLAS handle currently in use by a thread. The CUDA Fortran runtime keeps track of a CPU thread's current handle, if you are either using the legacy BLAS API, or do not wish to pass the handle through to low-level functions or subroutines manually.

```
type(cublashandle) function cublasGetHandle()
```

```
integer(4) function cublasGetHandle(handle)
  type(cublasHandle) :: handle
```

9.1.17. cublasSetVector

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy or array assignment statements.

```
integer(4) function cublassetvector(n, elemsize, x, incx, y, incy)
  integer :: n, elemsize, incx, incy
  integer*1, dimension(*) :: x
  integer*1, device, dimension(*) :: y
```

9.1.18. cublasGetVector

This function copies n elements from a vector x in GPU memory space to a vector y in host memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy or array assignment statements.

```
integer(4) function cublasgetvector(n, elemsize, x, incx, y, incy)
  integer :: n, elemsize, incx, incy
  integer*1, device, dimension(*) :: x
  integer*1, dimension(*) :: y
```

9.1.19. cublasSetMatrix

This function copies a tile of rows x cols elements from a matrix A in host memory space to a matrix B in GPU memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy, cudaMemcpy2D, or array assignment statements.

```
integer(4) function cublassetmatrix(rows, cols, elemsize, a, lda, b, ldb)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, dimension(lda, *) :: a
  integer*1, device, dimension(ldb, *) :: b
```

9.1.20. cublasGetMatrix

This function copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in host memory space. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpy, cudaMemcpy2D, or array assignment statements.

```
integer(4) function cublasgetmatrix(rows, cols, elemsize, a, lda, b, ldb)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, device, dimension(lda, *) :: a
  integer*1, dimension(ldb, *) :: b
```

9.1.21. cublasSetVectorAsync

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space, asynchronously, on the given CUDA stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync.

```
integer(4) function cublassetvectorasync(n, elemsize, x, incx, y, incy, stream)
  integer :: n, elemsize, incx, incy
  integer*1, dimension(*) :: x
  integer*1, device, dimension(*) :: y
  integer(kind=cuda_stream_kind()) :: stream
```

9.1.22. cublasGetVectorAsync

This function copies n elements from a vector x in host memory space to a vector y in GPU memory space, asynchronously, on the given CUDA stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of vector x and y is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync.

```
integer(4) function cublasgetvectorasync(n, elemsize, x, incx, y, incy, stream)
  integer :: n, elemsize, incx, incy
  integer*1, device, dimension(*) :: x
  integer*1, dimension(*) :: y
  integer(kind=cuda_stream_kind()) :: stream
```

9.1.23. cublasSetMatrixAsync

This function copies a tile of rows x cols elements from a matrix A in host memory space to a matrix B in GPU memory space, asynchronously using the specified stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync or cudaMemcpy2DAsync.

```
integer(4) function cublassetmatrixasync(rows, cols, elemsize, a, lda, b, ldb, stream)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, dimension(lda, *) :: a
  integer*1, device, dimension(ldb, *) :: b
  integer(kind=cuda_stream_kind()) :: stream
```

9.1.24. cublasGetMatrixAsync

This function copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in host memory space, asynchronously, using the specified stream. It is assumed that each element requires storage of elemSize bytes. In CUDA Fortran, the type of Matrix A and B is overloaded to take any data type, but the size of the data type must still be specified in bytes. This functionality can also be implemented using cudaMemcpyAsync or cudaMemcpy2DAsync.

```
integer(4) function cublasgetmatrixasync(rows, cols, elemsize, a, lda, b, ldb, stream)
  integer :: rows, cols, elemsize, lda, ldb
  integer*1, device, dimension(lda, *) :: a
  integer*1, dimension(ldb, *) :: b
  integer(kind=cuda_stream_kind()) :: stream
```

9.2. Single Precision Functions and Subroutines

This section contains interfaces to the single precision BLAS and cuBLAS functions and subroutines.

9.2.1. isamax

ISAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function isamax(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIsamax(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIsamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.2.2. isamin

ISAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function isamin(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIsamin(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIsamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.2.3. sasum

SASUM takes the sum of the absolute values.

```
real(4) function sasum(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasSasum(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.2.4. saxpy

SAXPY constant times a vector plus a vector.

```
subroutine saxpy(n, a, x, incx, y, incy)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSaxpy(n, a, x, incx, y, incy)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.2.5. scopy

SCOPY copies a vector, x, to a vector, y.

```
subroutine scopy(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasScopy(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasScopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.2.6. sdot

SDOT forms the dot product of two vectors.

```
real(4) function sdot(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
real(4) function cublasSdot(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSdot_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: res ! device or host variable
```

9.2.7. snrm2

SNRM2 returns the euclidean norm of a vector via the function name, so that SNRM2 := sqrt(x'*x).

```
real(4) function snrm2(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasSnrm2(n, x, incx)
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.2.8. srot

SROT applies a plane rotation.

```
subroutine srot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.2.9. srotg

SROTG constructs a Givens plane rotation.

```
subroutine srotg(sa, sb, sc, ss)
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

```
subroutine cublasSrotg(sa, sb, sc, ss)
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

```
integer(4) function cublasSrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

9.2.10. srotm

SROTM applies the modified Givens transformation, H, to the 2 by N matrix (SX**T) , where **T indicates transpose. The elements of SX are in (SX**T) SX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for SY using LY and INCY. With SPARAM(1)=SFLAG, H has one of the following forms.. SFLAG=-1.E0 SFLAG=0.E0 SFLAG=1.E0 SFLAG=-2.E0 (SH11 SH12) (1.E0 SH12) (SH11 1.E0) (1.E0 0.E0) H=( ) ( ) ( ) ( ) (SH21 SH22), (SH21 1.E0), (-1.E0 SH22), (0.E0 1.E0). See SROTMG for a description of data storage in SPARAM.

```
subroutine srotm(n, x, incx, y, incy, param)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSrotm(n, x, incx, y, incy, param)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: param(*) ! device or host variable
```

```
integer(4) function cublasSrotm_v2(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: param(*) ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.2.11. srotmg

SROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(SD1)*SX1,SQRT(SD2)*SY2)**T. With SPARAM(1)=SFLAG, H has one of the following forms..SFLAG=-1.E0 SFLAG=0.E0 SFLAG=1.E0 SFLAG=-2.E0 (SH11 SH12) (1.E0 SH12) (SH11 1.E0) (1.E0 0.E0) H=( ) ( ) ( ) ( ) (SH21 SH22), (SH21 1.E0), (-1.E0 SH22), (0.E0 1.E0). Locations 2-4 of SPARAM contain SH11,SH21,SH12, and SH22 respectively. (Values of 1.EO, -1.EO, or O.EO implied by the value of SPARAM(1) are not stored in SPARAM.)

```
subroutine srotmg(d1, d2, x1, y1, param)
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
subroutine cublasSrotmg(d1, d2, x1, y1, param)
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
integer(4) function cublasSrotmg_v2(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

9.2.12. sscal

SSCAL scales a vector by a constant.

```
subroutine sscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasSscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasSscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

9.2.13. sswap

SSWAP interchanges two vectors.

```
subroutine sswap(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasSswap(n, x, incx, y, incy)
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasSswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.2.14. sgbmv

SGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine sgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta,
↪y, incy)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.2.15. sgemv

SGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine sgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.2.16. sger

SGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine sger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSger_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

9.2.17. ssbmv

SSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals.

```
subroutine ssbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsbmv_v2(h, t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.2.18. sspmv

SSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSspmv_v2(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.2.19. sspr

SSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSspr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.2.20. sspr2

SSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine sspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSspr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

9.2.21. ssymv

SSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.2.22. ssyr

SSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine ssyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

9.2.23. ssyr2

SSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine ssyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x, y ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasSsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasSsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

9.2.24. stbmv

STBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine stbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.2.25. stbsv

STBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine stbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.2.26. stpmv

STPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine stpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasStpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasStpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

9.2.27. stpsv

STPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine stpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasStpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasStpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

9.2.28. strmv

STRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine strmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.2.29. strsv

STRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine strsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasStrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

```
integer(4) function cublasStrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.2.30. sgemm

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSgemm_v2(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
```

(continues on next page)

(continued from previous page)

```
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.31. ssymm

SSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.32. ssyrk

SSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.33. ssyr2k

SSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.34. ssyrkx

SSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine ssyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasSsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c

  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasSsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.35. strmm

STRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
subroutine strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasStrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasStrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha ! device or host variable
```

9.2.36. strsm

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
subroutine strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a ! device or host variable
  real(4), device, dimension(ldb, *) :: b ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasStrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasStrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

9.2.37. cublasSgeam

SGEAM performs the matrix-matrix addition/transposition C := alpha*op(A) + beta*op(B), where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, and A, B, and C are matrices with dimensions such that op(A), op(B), and C are each m by n.

```
integer(4) function cublasSgeam(h, transa, transb, m, n, alpha, a, lda, beta, b, ldb,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, lda, ldb, ldc
```

(continues on next page)

(continued from previous page)

```
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.2.38. cublasSgemvBatched

SGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasSgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasSgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

9.2.39. cublasSgemmBatched

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasSgemmBatched(h, transa, transb, m, n, k, alpha, Aarray, lda,
↪ Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasSgemmBatched_v2(h, transa, transb, m, n, k, alpha, &
           Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

9.2.40. cublasSgelsBatched

SGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasSgelsBatched(h, trans, m, n, nrhs, Aarray, lda, Carray, ldc,
↪ info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
```

(continues on next page)

(continued from previous page)

```
  integer, device :: devinfo(*)
  integer :: batchCount
```

9.2.41. cublasSgeqrfBatched

SGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R.

```
integer(4) function cublasSgeqrfBatched(h, m, n, Aarray, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

9.2.42. cublasSgetrfBatched

SGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasSgetrfBatched(h, n, Aarray, lda, ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

9.2.43. cublasSgetriBatched

SGETRI computes the inverse of a matrix using the LU factorization computed by SGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasSgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

9.2.44. cublasSgetrsBatched

SGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by SGETRF.

```
integer(4) function cublasSgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt, Barray,
↪ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

9.2.45. cublasSmatinvBatched

cublasSmatinvBatched is a short cut of cublasSgetrfBatched plus cublasSgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasSgetrfBatched and cublasSgetriBatched.

```
integer(4) function cublasSmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

9.2.46. cublasStrsmBatched

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasStrsmBatched( h, side, uplo, trans, diag, m, n, alpha, A,
↪lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasStrsmBatched_v2( h, side, uplo, trans, diag, m, n, alpha, A,
↪ lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

9.2.47. cublasSgemvStridedBatched

SGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasSgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(4), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(4), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
  real(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasSgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(4), device :: A(lda,*)
  integer :: lda
```

(continues on next page)

(continued from previous page)

```
  integer(8) :: strideA
  real(4), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
  real(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

9.2.48. cublasSgemmStridedBatched

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasSgemmStridedBatched(h, transa, transb, m, n, k, alpha,
↪Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  real(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(4), device :: beta ! device or host variable
  real(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasSgemmStridedBatched_v2(h, transa, transb, m, n, k, alpha, &
          Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
↪batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  real(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(4), device :: beta ! device or host variable
  real(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

9.3. Double Precision Functions and Subroutines

This section contains interfaces to the double precision BLAS and cuBLAS functions and subroutines.

9.3.1. idamax

IDAMAX finds the the index of the element having the maximum absolute value.

```
integer(4) function idamax(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIdamax(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIdamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.3.2. idamin

IDAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function idamin(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIdamin(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIdamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.3.3. dasum

DASUM takes the sum of the absolute values.

```
real(8) function dasum(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDasum(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.3.4. daxpy

DAXPY constant times a vector plus a vector.

```
subroutine daxpy(n, a, x, incx, y, incy)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDaxpy(n, a, x, incx, y, incy)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.3.5. dcopy

DCOPY copies a vector, x, to a vector, y.

```
subroutine dcopy(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDcopy(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.3.6. ddot

DDOT forms the dot product of two vectors.

```
real(8) function ddot(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
real(8) function cublasDdot(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDdot_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: res ! device or host variable
```

9.3.7. dnrm2

DNRM2 returns the euclidean norm of a vector via the function name, so that DNRM2 := sqrt(x'*x)

```
real(8) function dnrm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDnrm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.3.8. drot

DROT applies a plane rotation.

```
subroutine drot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.3.9. drotg

DROTG constructs a Givens plane rotation.

```
subroutine drotg(sa, sb, sc, ss)
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

```
subroutine cublasDrotg(sa, sb, sc, ss)
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

```
integer(4) function cublasDrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

9.3.10. drotm

DROTM applies the modified Givens transformation, H, to the 2 by N matrix (DX**T) , where **T indicates transpose. The elements of DX are in (DX**T) DX(LX+I*INCX), I = 0 to N-1, where LX = 1 if INCX .GE. 0, ELSE LX = (-INCX)*N, and similarly for DY using LY and INCY. With DPARAM(1)=DFLAG, H has one of the following forms.. DFLAG=-1.D0 DFLAG=0.D0 DFLAG=1.D0 DFLAG=-2.D0 (DH11 DH12) (1.D0 DH12) (DH11 1.D0) (1.D0 0.D0) H=( ) ( ) ( ) ( ) (DH21 DH22), (DH21 1.D0), (-1.D0 DH22), (0.D0 1.D0). See DROTMG for a description of data storage in DPARAM.

```
subroutine drotm(n, x, incx, y, incy, param)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDrotm(n, x, incx, y, incy, param)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: param(*) ! device or host variable
```

```
integer(4) function cublasDrotm_v2(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: param(*) ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.3.11. drotmg

DROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector (SQRT(DD1)*DX1,SQRT(DD2)*DY2)**T. With DPARAM(1)=DFLAG, H has one of the following forms.. DFLAG=-1.D0 DFLAG=0.D0 DFLAG=1.D0 DFLAG=-2.D0 (DH11 DH12) (1.D0 DH12) (DH11 1.D0) (1.D0 0.D0) H=( ) ( ) ( ) ( ) (DH21 DH22), (DH21 1.D0), (-1.D0 DH22), (0.D0 1.D0). Locations 2-4 of DPARAM contain DH11, DH21, DH12, and DH22 respectively. (Values of 1.D0, -1.D0, of 0.D0 implied by the value of DPARAM(1) are not stored in DPARAM.)

```
subroutine drotmg(d1, d2, x1, y1, param)
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
subroutine cublasDrotmg(d1, d2, x1, y1, param)
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

```
integer(4) function cublasDrotmg_v2(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

9.3.12. dscal

DSCAL scales a vector by a constant.

```
subroutine dscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasDscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

9.3.13. dswap

interchanges two vectors.

```
subroutine dswap(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasDswap(n, x, incx, y, incy)
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasDswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.3.14. dgbmv

DGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine dgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta,
↪y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.3.15. dgemv

DGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine dgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
```

(continues on next page)

(continued from previous page)

```
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.3.16. dger

DGER performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine dger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDger(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDger_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

9.3.17. dsbmv

DSBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric band matrix, with k super-diagonals.

```
subroutine dsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsbmv(t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsbmv_v2(h, t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.3.18. dspmv

DSPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDspmv(t, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDspmv_v2(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.3.19. dspr

DSPR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDspr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDspr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.3.20. dspr2

DSPR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix, supplied in packed form.

```
subroutine dspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDspr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDspr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

9.3.21. dsymv

DSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.3.22. dsyr

DSYR performs the symmetric rank 1 operation A := alpha*x*x**T + A, where alpha is a real scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine dsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

9.3.23. dsyr2

DSYR2 performs the symmetric rank 2 operation A := alpha*x*y**T + alpha*y*x**T + A, where alpha is a scalar, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine dsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x, y ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
```

(continues on next page)

(continued from previous page)

```
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

9.3.24. dtbmv

DTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine dtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.3.25. dtbsv

DTBSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine dtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.3.26. dtpmv

DTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine dtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasDtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasDtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

9.3.27. dtpsv

DTPSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine dtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasDtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasDtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

9.3.28. dtrmv

DTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine dtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.3.29. dtrsv

DTRSV solves one of the systems of equations A*x = b, or A**T*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine dtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasDtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

```
integer(4) function cublasDtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.3.30. dgemm

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDgemm_v2(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.31. dsymm

DSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.32. dsyrk

DSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.33. dsyr2k

DSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.34. dsyrkx

DSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine dsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasDsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
```

(continues on next page)

(continued from previous page)

```
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasDsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.35. dtrmm

DTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
subroutine dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha ! device or host variable
```

9.3.36. dtrsm

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
subroutine dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a ! device or host variable
  real(8), device, dimension(ldb, *) :: b ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasDtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasDtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

9.3.37. cublasDgeam

DGEAM performs the matrix-matrix addition/transposition C := alpha*op(A) + beta*op(B), where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, and A, B, and C are matrices with dimensions such that op(A), op(B), and C are each m by n.

```
integer(4) function cublasDgeam(h, transa, transb, m, n, alpha, a, lda, beta, b, ldb,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.3.38. cublasDgemvBatched

DGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasDgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasDgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

9.3.39. cublasDgemmBatched

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasDgemmBatched(h, transa, transb, m, n, k, alpha, Aarray, lda,
↪ Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
```

(continues on next page)

(continued from previous page)

```
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasDgemmBatched_v2(h, transa, transb, m, n, k, alpha, &
           Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

9.3.40. cublasDgelsBatched

DGELS solves overdetermined or underdetermined real linear systems involving an M-by-N matrix A, or its transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'T' and m >= n: find the minimum norm solution of an undetermined system A**T * X = B. 4. If TRANS = 'T' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**T * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasDgelsBatched(h, trans, m, n, nrhs, Aarray, lda, Carray, ldc,
↪ info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

9.3.41. cublasDgeqrfBatched

DGEQRF computes a QR factorization of a real M-by-N matrix A: A = Q * R.

```
integer(4) function cublasDgeqrfBatched(h, m, n, Aarray, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

9.3.42. cublasDgetrfBatched

DGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasDgetrfBatched(h, n, Aarray, lda, ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

9.3.43. cublasDgetriBatched

DGETRI computes the inverse of a matrix using the LU factorization computed by DGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasDgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

9.3.44. cublasDgetrsBatched

DGETRS solves a system of linear equations A * X = B or A**T * X = B with a general N-by-N matrix A using the LU factorization computed by DGETRF.

```
integer(4) function cublasDgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt, Barray,
↪ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

9.3.45. cublasDmatinvBatched

cublasDmatinvBatched is a short cut of cublasDgetrfBatched plus cublasDgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasDgetrfBatched and cublasDgetriBatched.

```
integer(4) function cublasDmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

9.3.46. cublasDtrsmBatched

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasDtrsmBatched( h, side, uplo, trans, diag, m, n, alpha, A,
↪lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  real(8), device :: alpha ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasDtrsmBatched_v2( h, side, uplo, trans, diag, m, n, alpha, A,
↪ lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

9.3.47. cublasDgemvStridedBatched

DGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasDgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  real(8), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(8), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(8), device :: beta ! device or host variable
  real(8), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasDgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(8), device :: alpha ! device or host variable
  real(8), device :: A(lda,*)
  integer :: lda
```

(continues on next page)

(continued from previous page)

```
  integer(8) :: strideA
  real(8), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(8), device :: beta ! device or host variable
  real(8), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

9.3.48. cublasDgemmStridedBatched

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasDgemmStridedBatched(h, transa, transb, m, n, k, alpha,
↪Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  real(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(8), device :: beta ! device or host variable
  real(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasDgemmStridedBatched_v2(h, transa, transb, m, n, k, alpha, &
          Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
↪batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(8), device :: alpha ! device or host variable
  real(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  real(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  real(8), device :: beta ! device or host variable
  real(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

9.4. Single Precision Complex Functions and Subroutines

This section contains interfaces to the single precision complex BLAS and cuBLAS functions and subroutines.

9.4.1. icamax

ICAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function icamax(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIcamax(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIcamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.4.2. icamin

ICAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function icamin(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIcamin(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIcamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.4.3. scasum

SCASUM takes the sum of the absolute values of a complex vector and returns a single precision result.

```
real(4) function scasum(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasScasum(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasScasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.4.4. caxpy

CAXPY constant times a vector plus a vector.

```
subroutine caxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.4.5. ccopy

CCOPY copies a vector x to a vector y.

```
subroutine ccopy(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCcopy(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.4.6. cdotc

forms the dot product of two vectors, conjugating the first vector.

```
complex(4) function cdotc(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(4) function cublasCdotc(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCdotc_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

9.4.7. cdotu

CDOTU forms the dot product of two vectors.

```
complex(4) function cdotu(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(4) function cublasCdotu(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCdotu_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

9.4.8. scnrm2

SCNRM2 returns the euclidean norm of a vector via the function name, so that SCNRM2 := sqrt(x**H*x)

```
real(4) function scnrm2(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(4) function cublasScnrm2(n, x, incx)
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasScnrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.4.9. crot

CROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex.

```
subroutine crot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.4.10. csrot

CSROT applies a plane rotation, where the cos and sin (c and s) are real and the vectors cx and cy are complex.

```
subroutine csrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCsrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.4.11. crotg

CROTG determines a complex Givens rotation.

```
subroutine crotg(sa, sb, sc, ss)
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

```
subroutine cublasCrotg(sa, sb, sc, ss)
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

```
integer(4) function cublasCrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

9.4.12. cscal

CSCAL scales a vector by a constant.

```
subroutine cscal(n, a, x, incx)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasCscal(n, a, x, incx)
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasCscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

9.4.13. csscal

CSSCAL scales a complex vector by a real constant.

```
subroutine csscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasCsscal(n, a, x, incx)
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasCsscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

9.4.14. cswap

CSWAP interchanges two vectors.

```
subroutine cswap(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasCswap(n, x, incx, y, incy)
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasCswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.4.15. cgbmv

CGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine cgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta,
↪y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.16. cgemv

CGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine cgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.17. cgerc

CGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine cgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCgerc_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.4.18. cgeru

CGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine cgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCgeru_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.4.19. csymv

CSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine csymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.20. csyr

CSYR performs the symmetric rank 1 operation A := alpha*x*x**H + A, where alpha is a complex scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine csyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

9.4.21. csyr2

CSYR2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a complex scalar, x and y are n element vectors and A is an n by n SY matrix.

```
subroutine csyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.4.22. ctbmv

CTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine ctbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.4.23. ctbsv

CTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ctbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.4.24. ctpmv

CTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine ctpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasCtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasCtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

9.4.25. ctpsv

CTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ctpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasCtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

```
integer(4) function cublasCtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

9.4.26. ctrmv

CTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine ctrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.4.27. ctrsv

CTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ctrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasCtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

```
integer(4) function cublasCtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.4.28. chbmv

CHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals.

```
subroutine chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChbmv_v2(h, uplo, n, k, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.29. chemv

CHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChemv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
```

(continues on next page)

(continued from previous page)

```
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.30. chpmv

CHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChpmv_v2(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.31. cher

CHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

```
subroutine cher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasCher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCher_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.4.32. cher2

CHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine cher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCher2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

9.4.33. chpr

CHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x ! device or host variable
  real(4), device :: alpha ! device or host variable
```

```
subroutine cublasChpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasChpr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.4.34. chpr2

CHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine chpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasChpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasChpr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

9.4.35. cgemm

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCgemm_v2(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
```

(continues on next page)

(continued from previous page)

```
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.36. csymm

CSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.37. csyrk

CSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.38. csyr2k

CSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.39. csyrkx

CSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine csyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.40. ctrmm

CTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
subroutine ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
```

9.4.41. ctrsm

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
subroutine ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device :: alpha ! device or host variable
```

```
subroutine cublasCtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

```
integer(4) function cublasCtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

9.4.42. chemm

CHEMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

```
subroutine chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasChemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasChemm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.43. cherk

CHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  real(4), device :: alpha, beta ! device or host variable
```

```
subroutine cublasCherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasCherk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.4.44. cher2k

CHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
subroutine cublasCher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
integer(4) function cublasCher2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

9.4.45. cherkx

CHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
subroutine cherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a ! device or host variable
  complex(4), device, dimension(ldb, *) :: b ! device or host variable
  complex(4), device, dimension(ldc, *) :: c ! device or host variable
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
subroutine cublasCherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
```

(continues on next page)

(continued from previous page)

```
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

```
integer(4) function cublasCherkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

9.4.46. cublasCgeam

CGEAM performs the matrix-matrix addition/transposition C := alpha*op(A) + beta*op(B), where op(A) is one of op(A) = A, op(A) = A**T or op(A) = A**H, alpha and beta are scalars, and A, B, and C are matrices with dimensions such that op(A), op(B), and C are each m by n.

```
integer(4) function cublasCgeam(h, transa, transb, m, n, alpha, a, lda, beta, b, ldb,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.4.47. cublasCgemvBatched

CGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasCgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasCgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

9.4.48. cublasCgemmBatched

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasCgemmBatched(h, transa, transb, m, n, k, alpha, Aarray, lda,
↪ Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasCgemmBatched_v2(h, transa, transb, m, n, k, alpha, &
          Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

9.4.49. cublasCgelsBatched

CGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasCgelsBatched(h, trans, m, n, nrhs, Aarray, lda, Carray, ldc,
↪ info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

9.4.50. cublasCgeqrfBatched

CGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

```
integer(4) function cublasCgeqrfBatched(h, m, n, Aarray, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

9.4.51. cublasCgetrfBatched

CGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasCgetrfBatched(h, n, Aarray, lda, ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

9.4.52. cublasCgetriBatched

CGETRI computes the inverse of a matrix using the LU factorization computed by CGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasCgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

9.4.53. cublasCgetrsBatched

CGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by CGETRF.

```
integer(4) function cublasCgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt, Barray,
↪ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

9.4.54. cublasCmatinvBatched

cublasCmatinvBatched is a short cut of cublasCgetrfBatched plus cublasCgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasCgetrfBatched and cublasCgetriBatched.

```
integer(4) function cublasCmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

9.4.55. cublasCtrsmBatched

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasCtrsmBatched( h, side, uplo, trans, diag, m, n, alpha, A,
↪lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasCtrsmBatched_v2( h, side, uplo, trans, diag, m, n, alpha, A,
↪ lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

9.4.56. cublasCgemvStridedBatched

CGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasCgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  complex(4), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasCgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  complex(4), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

9.4.57. cublasCgemmStridedBatched

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasCgemmStridedBatched(h, transa, transb, m, n, k, alpha,
↪Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
```

(continues on next page)

(continued from previous page)

```
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasCgemmStridedBatched_v2(h, transa, transb, m, n, k, alpha, &
           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
↪batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(4), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(4), device :: beta ! device or host variable
  complex(4), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

9.5. Double Precision Complex Functions and Subroutines

This section contains interfaces to the double precision complex BLAS and cuBLAS functions and subroutines.

9.5.1. izamax

IZAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function izamax(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIzamax(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIzamax_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.5.2. izamin

IZAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function izamin(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
integer(4) function cublasIzamin(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasIzamin_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.5.3. dzasum

DZASUM takes the sum of the absolute values.

```
real(8) function dzasum(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDzasum(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDzasum_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
```

(continues on next page)

(continued from previous page)

```
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.5.4. zaxpy

ZAXPY constant times a vector plus a vector.

```
subroutine zaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZaxpy(n, a, x, incx, y, incy)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.5.5. zcopy

ZCOPY copies a vector, x, to a vector, y.

```
subroutine zcopy(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZcopy(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZcopy_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.5.6. zdotc

ZDOTC forms the dot product of a vector.

```
complex(8) function zdotc(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(8) function cublasZdotc(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZdotc_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

9.5.7. zdotu

ZDOTU forms the dot product of two vectors.

```
complex(8) function zdotu(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
complex(8) function cublasZdotu(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZdotu_v2(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

9.5.8. dznrm2

DZNRM2 returns the euclidean norm of a vector via the function name, so that DZNRM2 := sqrt(x**H*x)

```
real(8) function dznrm2(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDznrm2(n, x, incx)
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDznrm2_v2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.5.9. zrot

ZROT applies a plane rotation, where the cos (C) is real and the sin (S) is complex, and the vectors CX and CY are complex.

```
subroutine zrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.5.10. zsrot

ZSROT applies a plane rotation, where the cos and sin (c and s) are real and the vectors cx and cy are complex.

```
subroutine zsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZsrot(n, x, incx, y, incy, sc, ss)
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZsrot_v2(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.5.11. zrotg

ZROTG determines a double complex Givens rotation.

```
subroutine zrotg(sa, sb, sc, ss)
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

```
subroutine cublasZrotg(sa, sb, sc, ss)
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

```
integer(4) function cublasZrotg_v2(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

9.5.12. zscal

ZSCAL scales a vector by a constant.

```
subroutine zscal(n, a, x, incx)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasZscal(n, a, x, incx)
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasZscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

9.5.13. zdscal

ZDSCAL scales a vector by a constant.

```
subroutine zdscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
subroutine cublasZdscal(n, a, x, incx)
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasZdscal_v2(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

9.5.14. zswap

ZSWAP interchanges two vectors.

```
subroutine zswap(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y ! device or host variable
  integer :: incx, incy
```

```
subroutine cublasZswap(n, x, incx, y, incy)
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

```
integer(4) function cublasZswap_v2(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.5.15. zgbmv

ZGBMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n band matrix, with kl sub-diagonals and ku super-diagonals.

```
subroutine zgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgbmv(t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgbmv_v2(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta,
↪y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.16. zgemv

ZGEMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, or y := alpha*A**H*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

```
subroutine zgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgemv(t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: t
```

(continues on next page)

(continued from previous page)

```
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgemv_v2(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.17. zgerc

ZGERC performs the rank 1 operation A := alpha*x*y**H + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine zgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZgerc(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZgerc_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.5.18. zgeru

ZGERU performs the rank 1 operation A := alpha*x*y**T + A, where alpha is a scalar, x is an m element vector, y is an n element vector and A is an m by n matrix.

```
subroutine zgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZgeru(m, n, alpha, x, incx, y, incy, a, lda)
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZgeru_v2(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.5.19. zsymv

ZSYMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n symmetric matrix.

```
subroutine zsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsymv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.20. zsyr

ZSYR performs the symmetric rank 1 operation A := alpha*x*x**H + A, where alpha is a complex scalar, x is an n element vector and A is an n by n symmetric matrix.

```
subroutine zsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZsyr(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZsyr_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

9.5.21. zsyr2

ZSYR2 performs the symmetric rank 2 operation A := alpha*x*y' + alpha*y*x' + A, where alpha is a complex scalar, x and y are n element vectors and A is an n by n SY matrix.

```
subroutine zsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZsyr2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZsyr2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.5.22. ztbmv

ZTBMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals.

```
subroutine ztbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
```

(continues on next page)

(continued from previous page)

```
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtbmv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtbmv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.5.23. ztbsv

ZTBSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular band matrix, with (k + 1) diagonals. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ztbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtbsv(u, t, d, n, k, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtbsv_v2(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.5.24. ztpmv

ZTPMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

```
subroutine ztpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasZtpmv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasZtpmv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

9.5.25. ztpsv

ZTPSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix, supplied in packed form. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ztpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
```

```
subroutine cublasZtpsv(u, t, d, n, a, x, incx)
  character*1 :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

```
integer(4) function cublasZtpsv_v2(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

9.5.26. ztrmv

ZTRMV performs one of the matrix-vector operations x := A*x, or x := A**T*x, or x := A**H*x, where x is an n element vector and A is an n by n unit, or non-unit, upper or lower triangular matrix.

```
subroutine ztrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtrmv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtrmv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.5.27. ztrsv

ZTRSV solves one of the systems of equations A*x = b, or A**T*x = b, or A**H*x = b, where b and x are n element vectors and A is an n by n unit, or non-unit, upper or lower triangular matrix. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
subroutine ztrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x ! device or host variable
```

```
subroutine cublasZtrsv(u, t, d, n, a, lda, x, incx)
  character*1 :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

```
integer(4) function cublasZtrsv_v2(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.5.28. zhbmv

ZHBMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian band matrix, with k super-diagonals.

```
subroutine zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhbmv_v2(h, uplo, n, k, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.29. zhemv

ZHEMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhemv_v2(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
```

(continues on next page)

(continued from previous page)

```
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.30. zhpmv

ZHPMV performs the matrix-vector operation y := alpha*A*x + beta*y, where alpha and beta are scalars, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhpmv(uplo, n, alpha, a, x, incx, beta, y, incy)
  character*1 :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhpmv_v2(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.31. zher

ZHER performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix.

```
subroutine zher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasZher(t, n, alpha, x, incx, a, lda)
  character*1 :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZher_v2(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.5.32. zher2

ZHER2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix.

```
subroutine zher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZher2(t, n, alpha, x, incx, y, incy, a, lda)
  character*1 :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZher2_v2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

9.5.33. zhpr

ZHPR performs the hermitian rank 1 operation A := alpha*x*x**H + A, where alpha is a real scalar, x is an n element vector and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x ! device or host variable
  real(8), device :: alpha ! device or host variable
```

```
subroutine cublasZhpr(t, n, alpha, x, incx, a)
  character*1 :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZhpr_v2(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.5.34. zhpr2

ZHPR2 performs the hermitian rank 2 operation A := alpha*x*y**H + conjg(alpha)*y*x**H + A, where alpha is a scalar, x and y are n element vectors and A is an n by n hermitian matrix, supplied in packed form.

```
subroutine zhpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZhpr2(t, n, alpha, x, incx, y, incy, a)
  character*1 :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZhpr2_v2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

9.5.35. zgemm

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine zgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZgemm_v2(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
```

(continues on next page)

(continued from previous page)

```
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.36. zsymm

ZSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
subroutine zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsymm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.37. zsyrk

ZSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyrk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.38. zsyr2k

ZSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyr2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.39. zsyrkx

ZSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
subroutine zsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZsyrkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZsyrkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.40. ztrmm

ZTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
subroutine ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZtrmm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
```

9.5.41. ztrsm

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
subroutine ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device :: alpha ! device or host variable
```

```
subroutine cublasZtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
  character*1 :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

```
integer(4) function cublasZtrsm_v2(h, side, uplo, transa, diag, m, n, alpha, a, lda,
↪b, ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

9.5.42. zhemm

ZHEMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

```
subroutine zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZhemm_v2(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.43. zherk

ZHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
subroutine zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  real(8), device :: alpha, beta ! device or host variable
```

```
subroutine cublasZherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

```
integer(4) function cublasZherk_v2(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.5.44. zher2k

ZHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
subroutine zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
subroutine cublasZher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
integer(4) function cublasZher2k_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

9.5.45. zherkx

ZHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
subroutine zherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a ! device or host variable
  complex(8), device, dimension(ldb, *) :: b ! device or host variable
  complex(8), device, dimension(ldc, *) :: c ! device or host variable
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
subroutine cublasZherkx(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
  character*1 :: uplo, trans
```

(continues on next page)

(continued from previous page)

```
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

```
integer(4) function cublasZherkx_v2(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪ c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

9.5.46. cublasZgeam

ZGEAM performs the matrix-matrix addition/transposition C := alpha*op(A) + beta*op(B), where op(A) is one of op(A) = A, op(A) = A**T or op(A) = A**H, alpha and beta are scalars, and A, B, and C are matrices with dimensions such that op(A), op(B), and C are each m by n.

```
integer(4) function cublasZgeam(h, transa, transb, m, n, alpha, a, lda, beta, b, ldb,
↪c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.5.47. cublasZgemvBatched

ZGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasZgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
```

(continues on next page)

(continued from previous page)

```
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasZgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

9.5.48. cublasZgemmBatched

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasZgemmBatched(h, transa, transb, m, n, k, alpha, Aarray, lda,
↪ Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasZgemmBatched_v2(h, transa, transb, m, n, k, alpha, &
          Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
```

(continues on next page)

(continued from previous page)

```
  complex(8), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

9.5.49. cublasZgelsBatched

ZGELS solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix A, or its conjugate-transpose, using a QR or LQ factorization of A. It is assumed that A has full rank. The following options are provided: 1. If TRANS = 'N' and m >= n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A*X ||. 2. If TRANS = 'N' and m < n: find the minimum norm solution of an underdetermined system A * X = B. 3. If TRANS = 'C' and m >= n: find the minimum norm solution of an undetermined system A**H * X = B. 4. If TRANS = 'C' and m < n: find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize || B - A**H * X ||. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the M-by-NRHS right hand side matrix B and the N-by-NRHS solution matrix X.

```
integer(4) function cublasZgelsBatched(h, trans, m, n, nrhs, Aarray, lda, Carray, ldc,
↪ info, devinfo, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

9.5.50. cublasZgeqrfBatched

ZGEQRF computes a QR factorization of a complex M-by-N matrix A: A = Q * R.

```
integer(4) function cublasZgeqrfBatched(h, m, n, Aarray, lda, Tau, info, batchCount)
  type(cublasHandle) :: h
  integer :: m, n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

9.5.51. cublasZgetrfBatched

ZGETRF computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges. The factorization has the form A = P * L * U where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). This is the right-looking Level 3 BLAS version of the algorithm.

```
integer(4) function cublasZgetrfBatched(h, n, Aarray, lda, ipvt, info, batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

9.5.52. cublasZgetriBatched

ZGETRI computes the inverse of a matrix using the LU factorization computed by ZGETRF. This method inverts U and then computes inv(A) by solving the system inv(A)*L = inv(U) for inv(A).

```
integer(4) function cublasZgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchCount
```

9.5.53. cublasZgetrsBatched

ZGETRS solves a system of linear equations A * X = B, A**T * X = B, or A**H * X = B with a general N-by-N matrix A using the LU factorization computed by ZGETRF.

```
integer(4) function cublasZgetrsBatched(h, trans, n, nrhs, Aarray, lda, ipvt, Barray,
↪ldb, info, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: n, nrhs
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

9.5.54. cublasZmatinvBatched

cublasZmatinvBatched is a short cut of cublasZgetrfBatched plus cublasZgetriBatched. However it only works if n is less than 32. If not, the user has to go through cublasZgetrfBatched and cublasZgetriBatched.

```
integer(4) function cublasZmatinvBatched(h, n, Aarray, lda, Ainv, lda_inv, info,
↪batchCount)
  type(cublasHandle) :: h
  integer :: n
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

9.5.55. cublasZtrsmBatched

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasZtrsmBatched( h, side, uplo, trans, diag, m, n, alpha, A,
↪lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side ! integer or character(1) variable
  integer :: uplo ! integer or character(1) variable
  integer :: trans ! integer or character(1) variable
  integer :: diag ! integer or character(1) variable
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

```
integer(4) function cublasZtrsmBatched_v2( h, side, uplo, trans, diag, m, n, alpha, A,
↪ lda, B, ldb, batchCount)
  type(cublasHandle) :: h
  integer :: side
  integer :: uplo
  integer :: trans
  integer :: diag
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  type(c_devptr), device :: A(*)
  integer :: lda
  type(c_devptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

9.5.56. cublasZgemvStridedBatched

ZGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

```
integer(4) function cublasZgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  complex(8), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasZgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  complex(8), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

9.5.57. cublasZgemmStridedBatched

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasZgemmStridedBatched(h, transa, transb, m, n, k, alpha,
↪Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
```

(continues on next page)

(continued from previous page)

```
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

```
integer(4) function cublasZgemmStridedBatched_v2(h, transa, transb, m, n, k, alpha, &
           Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc, strideC,
↪batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: Aarray(*)
  integer :: lda
  integer :: strideA
  complex(8), device :: Barray(*)
  integer :: ldb
  integer :: strideB
  complex(8), device :: beta ! device or host variable
  complex(8), device :: Carray(*)
  integer :: ldc
  integer :: strideC
  integer :: batchCount
```

9.6. Half Precision Functions and Extension Functions

This section contains interfaces to the half precision cuBLAS functions and the BLAS extension functions which allow the user to individually specify the types of the arrays and computation (many or all of which support half precision).

The extension functions can accept one of many supported datatypes. Users should always check the latest cuBLAS documentation for supported combinations. In this document we will use the real(2) datatype since those functions are not otherwise supported by the `S`, `D`, `C`, and `Z` variants in the libraries. In addition, the user is responsible for properly setting the pointer mode by making calls to `cublasSetPointerMode` for all extension functions.

The `type(cudaDataType)` is now common to several of the newer library functions covered in this document. Though some functions will accept an appropriately valued integer, the use of `type(cudaDataType)` is now recommended going forward.

9.6.1. cublasHgemvBatched

HGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

In the HSH versions, `alpha, beta` are real(4), and the arrays which are pointed to should all contain real(2) data.

```
integer(4) function cublasHSHgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasHSHgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

In the HSS versions, `alpha, beta` are real(4), the `Aarray, xarray` arrays which are pointed to should contain real(2) data, and `yarray` should contain real(4) data.

```
integer(4) function cublasHSSgemvBatched(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

```
integer(4) function cublasHSSgemvBatched_v2(h, trans, m, n, alpha, &
      Aarray, lda, xarray, incx, beta, yarray, incy, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devptr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

9.6.2. cublasHgemvStridedBatched

HGEMV performs a batch of the matrix-vector operations Y := alpha*op(A) * X + beta*Y, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars, A is an m by n matrix, and X and Y are vectors.

In the HSH versions, `alpha, beta` are real(4), and the arrays `A, X, Y` are all real(2) data.

```
integer(4) function cublasHSHgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
  real(2), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasHSHgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  real(2), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

In the HSS versions, `alpha, beta` are real(4), the `A, X` arrays contain real(2) data, and the `Y` array contains real(4) data.

```
integer(4) function cublasHSSgemvStridedBatched(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
  real(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

```
integer(4) function cublasHSSgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchCount)
  type(cublasHandle) :: h
  integer :: trans
  integer :: m, n
  real(4), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: X(*)
  integer :: incx
  integer(8) :: strideX
  real(4), device :: beta ! device or host variable
  real(4), device :: Y(*)
  integer :: incy
  integer(8) :: strideY
  integer :: batchCount
```

9.6.3. cublasHgemm

HGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
subroutine cublasHgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, &
      beta, c, ldc)
  integer :: transa  ! integer or character(1) variable
```

(continues on next page)

(continued from previous page)

```
  integer :: transb  ! integer or character(1) variable
  integer :: m, n, k, lda, ldb, ldc
  real(2), device, dimension(lda, *) :: a
  real(2), device, dimension(ldb, *) :: b
  real(2), device, dimension(ldc, *) :: c
  real(2), device :: alpha, beta ! device or host variable
```

In the v2 version, the user is responsible for setting the pointer mode for the `alpha, beta` arguments.

```
integer(4) function cublasHgemm_v2(h, transa, transb, m, n, k, alpha, &
      a, lda, b, ldb, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(2), device, dimension(lda, *) :: a
  real(2), device, dimension(ldb, *) :: b
  real(2), device, dimension(ldc, *) :: c
  real(2), device :: alpha, beta ! device or host variable
```

9.6.4. cublasHgemmBatched

HGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasHgemmBatched(h, transa, transb, m, n, k, &
      alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(2), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

```
integer(4) function cublasHgemmBatched_v2(h, transa, transb, m, n, k, &
      alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  integer :: lda
  type(c_devptr), device :: Barray(*)
  integer :: ldb
  real(2), device :: beta ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  type(c_devptr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount
```

9.6.5. cublasHgemmStridedBatched

HGEMM performs a set of matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasHgemmStridedBatched(h, transa, transb, m, n, k, &
    alpha, A, lda, strideA, B, ldb, strideB, beta, C, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa ! integer or character(1) variable
  integer :: transb ! integer or character(1) variable
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: B(ldb,*)
  integer :: ldb
  integer(8) :: strideB
  real(2), device :: beta ! device or host variable
  real(2), device :: C(ldc,*)
  integer :: ldc
  integer(8) :: strideC
  integer :: batchCount
```

```
integer(4) function cublasHgemmStridedBatched_v2(h, transa, transb, m, n, k, &
    alpha, A, lda, strideA, B, ldb, strideB, beta, C, ldc, strideC, batchCount)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  integer :: lda
  integer(8) :: strideA
  real(2), device :: B(ldb,*)
  integer :: ldb
  integer(8) :: strideB
  real(2), device :: beta ! device or host variable
  real(2), device :: C(ldc,*)
  integer :: ldc
  integer(8) :: strideC
  integer :: batchCount
```

9.6.6. cublasIamaxEx

IAMAX finds the index of the element having the maximum absolute value.

```
integer(4) function cublasIamaxEx(h, n, x, xtype, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  integer, device :: res ! device or host variable
```

9.6.7. cublasIaminEx

IAMIN finds the index of the element having the minimum absolute value.

```
integer(4) function cublasIaminEx(h, n, x, xtype, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  integer, device :: res ! device or host variable
```

9.6.8. cublasAsumEx

ASUM takes the sum of the absolute values.

```
integer(4) function cublasAsumEx(h, n, x, xtype, incx, res, &
      restype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device :: res ! device or host variable
  type(cudaDataType) :: restype
  type(cudaDataType) :: extype
```

9.6.9. cublasAxpyEx

AXPY computes a constant times a vector plus a vector.

```
integer(4) function cublasAxpyEx(h, n, alpha, alphatype, &
      x, xtype, incx, y, ytype, incy, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device :: alpha
```

(continues on next page)

(continued from previous page)

```
  type(cudaDataType) :: alphatype
  real(2), device, dimension(*) :: x
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y
  type(cudaDataType) :: ytype
  integer :: incy
  type(cudaDataType) :: extype
```

9.6.10. cublasCopyEx

COPY copies a vector, x, to a vector, y.

```
integer(4) function cublasCopyEx(h, n, x, xtype, incx, &
      y, ytype, incy)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y
  type(cudaDataType) :: ytype
  integer :: incy
```

9.6.11. cublasDotEx

DOT forms the dot product of two vectors.

```
integer(4) function cublasDotEx(h, n, x, xtype, incx, &
      y, ytype, incy, res, restype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y  ! Type and kind as specified by ytype
  type(cudaDataType) :: ytype
  integer :: incy
  real(2), device :: res ! device or host variable
  type(cudaDataType) :: restype
  type(cudaDataType) :: extype
```

9.6.12. cublasDotcEx

DOTC forms the conjugated dot product of two vectors.

```
integer(4) function cublasDotcEx(h, n, x, xtype, incx, &
      y, ytype, incy, res, restype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y  ! Type and kind as specified by ytype
  type(cudaDataType) :: ytype
  integer :: incy
  real(2), device :: res ! device or host variable
  type(cudaDataType) :: restype
  type(cudaDataType) :: extype
```

9.6.13. cublasNrm2Ex

NRM2 produces the euclidean norm of a vector.

```
integer(4) function cublasNrm2Ex(h, n, x, xtype, incx, res, &
      restype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device :: res ! device or host variable
  type(cudaDataType) :: restype
  type(cudaDataType) :: extype
```

9.6.14. cublasRotEx

ROT applies a plane rotation.

```
integer(4) function cublasRotEx(h, n, x, xtype, incx, &
      y, ytype, incy, c, s, cstype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y  ! Type and kind as specified by ytype
  type(cudaDataType) :: ytype
  integer :: incy
  real(2), device :: c, s ! device or host variable
  type(cudaDataType) :: cstype
  type(cudaDataType) :: extype
```

9.6.15. cublasRotgEx

ROTG constructs a Givens plane rotation

```
integer(4) function cublasRotgEx(h, a, b, abtype, &
      c, s, cstype, extype)
  type(cublasHandle) :: h
  real(2), device :: a, b  ! Type and kind as specified by abtype
  type(cudaDataType) :: abtype
  real(2), device :: c, s ! device or host variable
  type(cudaDataType) :: cstype
  type(cudaDataType) :: extype
```

9.6.16. cublasRotmEx

ROTM applies a modified Givens transformation.

```
integer(4) function cublasRotmEx(h, n, x, xtype, incx, &
      y, ytype, incy, param, paramtype, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x  ! Type and kind as specified by xtype
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y  ! Type and kind as specified by ytype
  type(cudaDataType) :: ytype
  integer :: incy
  real(2), device, dimension(*) :: param
  type(cudaDataType) :: paramtype
  type(cudaDataType) :: extype
```

9.6.17. cublasRotmgEx

ROTMG constructs a modified Givens transformation matrix.

```
integer(4) function cublasRotmgEx(h, d1, d1type, d2, d2type, &
      x1, x1type, y1, y1type, param, paramtype, extype)
  type(cublasHandle) :: h
  real(2), device :: d1  ! Type and kind as specified by d1type
  type(cudaDataType) :: d1type
  real(2), device :: d2  ! Type and kind as specified by d2type
  type(cudaDataType) :: d2type
  real(2), device :: x1  ! Type and kind as specified by x1type
  type(cudaDataType) :: x1type
  real(2), device :: y1  ! Type and kind as specified by y1type
  type(cudaDataType) :: y1type
  real(2), device, dimension(*) :: param
  type(cudaDataType) :: paramtype
  type(cudaDataType) :: extype
```

9.6.18. cublasScalEx

SCAL scales a vector by a constant.

```
integer(4) function cublasScalEx(h, n, alpha, alphatype, &
      x, xtype, incx, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device :: alpha
  type(cudaDataType) :: alphatype
  real(2), device, dimension(*) :: x
  type(cudaDataType) :: xtype
  integer :: incx
  type(cudaDataType) :: extype
```

9.6.19. cublasSwapEx

SWAP interchanges two vectors.

```
integer(4) function cublasSwapEx(h, n, x, xtype, incx, &
      y, ytype, incy)
  type(cublasHandle) :: h
  integer :: n
  real(2), device, dimension(*) :: x
  type(cudaDataType) :: xtype
  integer :: incx
  real(2), device, dimension(*) :: y
  type(cudaDataType) :: ytype
  integer :: incy
```

9.6.20. cublasGemmEx

GEMM performs the matrix-matrix multiply operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix, and C an m by n matrix.

The data type of `alpha, beta` mainly follows the computeType argument. See the cuBLAS documentation for data type combinations currently supported.

```
integer(4) function cublasGemmEx(h, transa, transb, m, n, k, alpha, &
      A, atype, lda, B, btype, ldb, beta, C, ctype, ldc, computeType, algo)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  type(cudaDataType) :: atype
  integer :: lda
  real(2), device :: B(ldb,*)
  type(cudaDataType) :: btype
  integer :: ldb
```

(continues on next page)

(continued from previous page)

```
  real(2), device :: beta ! device or host variable
  real(2), device :: C(ldc,*)
  type(cudaDataType) :: ctype
  integer :: ldc
  type(cublasComputeType) :: computeType  ! also accept integer
  type(cublasGemmAlgoType) :: algo        ! also accept integer
```

9.6.21. cublasGemmBatchedEx

GEMM performs a batch of matrix-matrix multiply operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix, and C an m by n matrix.

The data type of `alpha, beta` mainly follows the computeType argument. See the cuBLAS documentation for data type combinations currently supported.

```
integer(4) function cublasGemmBatchedEx(h, transa, transb, m, n, k, &
      alpha, Aarray, atype, lda, Barray, btype, ldb, beta, &
      Carray, ctype, ldc, batchCount, computeType, algo)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  type(c_devptr), device :: Aarray(*)
  type(cudaDataType) :: atype
  integer :: lda
  type(c_devptr), device :: Barray(*)
  type(cudaDataType) :: btype
  integer :: ldb
  real(2), device :: beta ! device or host variable
  type(c_devptr), device :: Carray(*)
  type(cudaDataType) :: ctype
  integer :: ldc
  integer :: batchCount
  type(cublasComputeType) :: computeType  ! also accept integer
  type(cublasGemmAlgoType) :: algo        ! also accept integer
```

9.6.22. cublasGemmStridedBatchedEx

GEMM performs a batch of matrix-matrix multiply operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix, and C an m by n matrix.

The data type of `alpha, beta` mainly follows the computeType argument. See the cuBLAS documentation for data type combinations currently supported.

```
integer(4) function cublasGemmStridedBatchedEx(h, transa, transb, m, n, k, &
      alpha, A, atype, lda, strideA, B, btype, ldb, strideB, beta, &
      C, ctype, ldc, strideC, batchCount, computeType, algo)
  type(cublasHandle) :: h
  integer :: transa, transb
```

(continues on next page)

(continued from previous page)

```
  integer :: m, n, k
  real(2), device :: alpha ! device or host variable
  real(2), device :: A(lda,*)
  type(cudaDataType) :: atype
  integer :: lda
  integer(8) :: strideA
  real(2), device :: B(ldb,*)
  type(cudaDataType) :: btype
  integer :: ldb
  integer(8) :: strideB
  real(2), device :: beta ! device or host variable
  real(2), device :: C(ldc,*)
  type(cudaDataType) :: ctype
  integer :: ldc
  integer(8) :: strideC
  integer :: batchCount
  type(cublasComputeType) :: computeType  ! also accept integer
  type(cublasGemmAlgoType) :: algo        ! also accept integer
```

9.7. CUBLAS V2 Module Functions

This section contains interfaces to the cuBLAS V2 Module Functions. Users can access this module by inserting the line `use cublas_v2` into the program unit. One major difference in the `cublas_v2` versus the `cublas` module is the cublas entry points, such as `cublasIsamax` are changed to take the handle as the first argument. The second difference in the `cublas_v2` module is the v2 entry points, such as `cublasIsamax_v2` do not implicitly handle the pointer modes for the user. It is up to the programmer to make calls to `cublasSetPointerMode` to tell the library if scalar arguments reside on the host or device. The actual interfaces to the v2 entry points do not change, and are not listed in this section.

9.7.1. Single Precision Functions and Subroutines

This section contains the V2 interfaces to the single precision BLAS and cuBLAS functions and subroutines.

9.7.1.1 isamax

If you use the cublas_v2 module, the interface for cublasIsamax is changed to the following:

```
integer(4) function cublasIsamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.1.2 isamin

If you use the cublas_v2 module, the interface for cublasIsamin is changed to the following:

```
integer(4) function cublasIsamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.1.3 sasum

If you use the cublas_v2 module, the interface for cublasSasum is changed to the following:

```
integer(4) function cublasSasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.7.1.4 saxpy

If you use the cublas_v2 module, the interface for cublasSaxpy is changed to the following:

```
integer(4) function cublasSaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.1.5 scopy

If you use the cublas_v2 module, the interface for cublasScopy is changed to the following:

```
integer(4) function cublasScopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.1.6 sdot

If you use the cublas_v2 module, the interface for cublasSdot is changed to the following:

```
integer(4) function cublasSdot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
  real(4), device :: res ! device or host variable
```

9.7.1.7 snrm2

If you use the cublas_v2 module, the interface for cublasSnrm2 is changed to the following:

```
integer(4) function cublasSnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.7.1.8 srot

If you use the cublas_v2 module, the interface for cublasSrot is changed to the following:

```
integer(4) function cublasSrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.1.9 srotg

If you use the cublas_v2 module, the interface for cublasSrotg is changed to the following:

```
integer(4) function cublasSrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(4), device :: sa, sb, sc, ss ! device or host variable
```

9.7.1.10 srotm

If you use the cublas_v2 module, the interface for cublasSrotm is changed to the following:

```
integer(4) function cublasSrotm(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: param(*) ! device or host variable
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.1.11 srotmg

If you use the cublas_v2 module, the interface for cublasSrotmg is changed to the following:

```
integer(4) function cublasSrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(4), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

9.7.1.12 sscal

If you use the cublas_v2 module, the interface for cublasSscal is changed to the following:

```
integer(4) function cublasSscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  real(4), device, dimension(*) :: x
  integer :: incx
```

9.7.1.13 sswap

If you use the cublas_v2 module, the interface for cublasSswap is changed to the following:

```
integer(4) function cublasSswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.1.14 sgbmv

If you use the cublas_v2 module, the interface for cublasSgbmv is changed to the following:

```
integer(4) function cublasSgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.15 sgemv

If you use the cublas_v2 module, the interface for cublasSgemv is changed to the following:

```
integer(4) function cublasSgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.16 sger

If you use the cublas_v2 module, the interface for cublasSger is changed to the following:

```
integer(4) function cublasSger(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

9.7.1.17 ssbmv

If you use the cublas_v2 module, the interface for cublasSsbmv is changed to the following:

```
integer(4) function cublasSsbmv(h, t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.18 sspmv

If you use the cublas_v2 module, the interface for cublasSspmv is changed to the following:

```
integer(4) function cublasSspmv(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.19 sspr

If you use the cublas_v2 module, the interface for cublasSspr is changed to the following:

```
integer(4) function cublasSspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.7.1.20 sspr2

If you use the cublas_v2 module, the interface for cublasSspr2 is changed to the following:

```
integer(4) function cublasSspr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

9.7.1.21 ssymv

If you use the cublas_v2 module, the interface for cublasSsymv is changed to the following:

```
integer(4) function cublasSsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.22 ssyr

If you use the cublas_v2 module, the interface for cublasSsyr is changed to the following:

```
integer(4) function cublasSsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
  real(4), device :: alpha ! device or host variable
```

9.7.1.23 ssyr2

If you use the cublas_v2 module, the interface for cublasSsyr2 is changed to the following:

```
integer(4) function cublasSsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x, y
  real(4), device :: alpha ! device or host variable
```

9.7.1.24 stbmv

If you use the cublas_v2 module, the interface for cublasStbmv is changed to the following:

```
integer(4) function cublasStbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.7.1.25 stbsv

If you use the cublas_v2 module, the interface for cublasStbsv is changed to the following:

```
integer(4) function cublasStbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.7.1.26 stpmv

If you use the cublas_v2 module, the interface for cublasStpmv is changed to the following:

```
integer(4) function cublasStpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

9.7.1.27 stpsv

If you use the cublas_v2 module, the interface for cublasStpsv is changed to the following:

```
integer(4) function cublasStpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x
```

9.7.1.28 strmv

If you use the cublas_v2 module, the interface for cublasStrmv is changed to the following:

```
integer(4) function cublasStrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.7.1.29 strsv

If you use the cublas_v2 module, the interface for cublasStrsv is changed to the following:

```
integer(4) function cublasStrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(*) :: x
```

9.7.1.30 sgemm

If you use the cublas_v2 module, the interface for cublasSgemm is changed to the following:

```
integer(4) function cublasSgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.31 ssymm

If you use the cublas_v2 module, the interface for cublasSsymm is changed to the following:

```
integer(4) function cublasSsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.32 ssyrk

If you use the cublas_v2 module, the interface for cublasSsyrk is changed to the following:

```
integer(4) function cublasSsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.33 ssyr2k

If you use the cublas_v2 module, the interface for cublasSsyr2k is changed to the following:

```
integer(4) function cublasSsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.34 ssyrkx

If you use the cublas_v2 module, the interface for cublasSsyrkx is changed to the following:

```
integer(4) function cublasSsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.1.35 strmm

If you use the cublas_v2 module, the interface for cublasStrmm is changed to the following:

```
integer(4) function cublasStrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha ! device or host variable
```

9.7.1.36 strsm

If you use the cublas_v2 module, the interface for cublasStrsm is changed to the following:

```
integer(4) function cublasStrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(4), device, dimension(lda, *) :: a
  real(4), device, dimension(ldb, *) :: b
  real(4), device :: alpha ! device or host variable
```

9.7.2. Double Precision Functions and Subroutines

This section contains the V2 interfaces to the double precision BLAS and cuBLAS functions and subroutines.

9.7.2.1 idamax

If you use the cublas_v2 module, the interface for cublasIdamax is changed to the following:

```
integer(4) function cublasIdamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.2.2 idamin

If you use the cublas_v2 module, the interface for cublasIdamin is changed to the following:

```
integer(4) function cublasIdamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.2.3 dasum

If you use the cublas_v2 module, the interface for cublasDasum is changed to the following:

```
integer(4) function cublasDasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.7.2.4 daxpy

If you use the cublas_v2 module, the interface for cublasDaxpy is changed to the following:

```
integer(4) function cublasDaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.2.5 dcopy

If you use the cublas_v2 module, the interface for cublasDcopy is changed to the following:

```
integer(4) function cublasDcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.2.6 ddot

If you use the cublas_v2 module, the interface for cublasDdot is changed to the following:

```
integer(4) function cublasDdot(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
  real(8), device :: res ! device or host variable
```

9.7.2.7 dnrm2

If you use the cublas_v2 module, the interface for cublasDnrm2 is changed to the following:

```
integer(4) function cublasDnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.7.2.8 drot

If you use the cublas_v2 module, the interface for cublasDrot is changed to the following:

```
integer(4) function cublasDrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.2.9 drotg

If you use the cublas_v2 module, the interface for cublasDrotg is changed to the following:

```
integer(4) function cublasDrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  real(8), device :: sa, sb, sc, ss ! device or host variable
```

9.7.2.10 drotm

If you use the cublas_v2 module, the interface for cublasDrotm is changed to the following:

```
integer(4) function cublasDrotm(h, n, x, incx, y, incy, param)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: param(*) ! device or host variable
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.2.11 drotmg

If you use the cublas_v2 module, the interface for cublasDrotmg is changed to the following:

```
integer(4) function cublasDrotmg(h, d1, d2, x1, y1, param)
  type(cublasHandle) :: h
  real(8), device :: d1, d2, x1, y1, param(*) ! device or host variable
```

9.7.2.12 dscal

If you use the cublas_v2 module, the interface for cublasDscal is changed to the following:

```
integer(4) function cublasDscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  real(8), device, dimension(*) :: x
  integer :: incx
```

9.7.2.13 dswap

If you use the cublas_v2 module, the interface for cublasDswap is changed to the following:

```
integer(4) function cublasDswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.2.14 dgbmv

If you use the cublas_v2 module, the interface for cublasDgbmv is changed to the following:

```
integer(4) function cublasDgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.15 dgemv

If you use the cublas_v2 module, the interface for cublasDgemv is changed to the following:

```
integer(4) function cublasDgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.16 dger

If you use the cublas_v2 module, the interface for cublasDger is changed to the following:

```
integer(4) function cublasDger(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

9.7.2.17 dsbmv

If you use the cublas_v2 module, the interface for cublasDsbmv is changed to the following:

```
integer(4) function cublasDsbmv(h, t, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: k, n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.18 dspmv

If you use the cublas_v2 module, the interface for cublasDspmv is changed to the following:

```
integer(4) function cublasDspmv(h, t, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.19 dspr

If you use the cublas_v2 module, the interface for cublasDspr is changed to the following:

```
integer(4) function cublasDspr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.7.2.20 dspr2

If you use the cublas_v2 module, the interface for cublasDspr2 is changed to the following:

```
integer(4) function cublasDspr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  real(8), device, dimension(*) :: a, x, y
  real(8), device :: alpha ! device or host variable
```

9.7.2.21 dsymv

If you use the cublas_v2 module, the interface for cublasDsymv is changed to the following:

```
integer(4) function cublasDsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.22 dsyr

If you use the cublas_v2 module, the interface for cublasDsyr is changed to the following:

```
integer(4) function cublasDsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
  real(8), device :: alpha ! device or host variable
```

9.7.2.23 dsyr2

If you use the cublas_v2 module, the interface for cublasDsyr2 is changed to the following:

```
integer(4) function cublasDsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha ! device or host variable
```

9.7.2.24 dtbmv

If you use the cublas_v2 module, the interface for cublasDtbmv is changed to the following:

```
integer(4) function cublasDtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.7.2.25 dtbsv

If you use the cublas_v2 module, the interface for cublasDtbsv is changed to the following:

```
integer(4) function cublasDtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.7.2.26 dtpmv

If you use the cublas_v2 module, the interface for cublasDtpmv is changed to the following:

```
integer(4) function cublasDtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

9.7.2.27 dtpsv

If you use the cublas_v2 module, the interface for cublasDtpsv is changed to the following:

```
integer(4) function cublasDtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
```

9.7.2.28 dtrmv

If you use the cublas_v2 module, the interface for cublasDtrmv is changed to the following:

```
integer(4) function cublasDtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.7.2.29 dtrsv

If you use the cublas_v2 module, the interface for cublasDtrsv is changed to the following:

```
integer(4) function cublasDtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

9.7.2.30 dgemm

If you use the cublas_v2 module, the interface for cublasDgemm is changed to the following:

```
integer(4) function cublasDgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.31 dsymm

If you use the cublas_v2 module, the interface for cublasDsymm is changed to the following:

```
integer(4) function cublasDsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.32 dsyrk

If you use the cublas_v2 module, the interface for cublasDsyrk is changed to the following:

```
integer(4) function cublasDsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.33 dsyr2k

If you use the cublas_v2 module, the interface for cublasDsyr2k is changed to the following:

```
integer(4) function cublasDsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.34 dsyrkx

If you use the cublas_v2 module, the interface for cublasDsyrkx is changed to the following:

```
integer(4) function cublasDsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.2.35 dtrmm

If you use the cublas_v2 module, the interface for cublasDtrmm is changed to the following:

```
integer(4) function cublasDtrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha ! device or host variable
```

9.7.2.36 dtrsm

If you use the cublas_v2 module, the interface for cublasDtrsm is changed to the following:

```
integer(4) function cublasDtrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(ldb, *) :: b
  real(8), device :: alpha ! device or host variable
```

9.7.3. Single Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the single precision complex BLAS and cuBLAS functions and subroutines.

9.7.3.1 icamax

If you use the cublas_v2 module, the interface for cublasIcamax is changed to the following:

```
integer(4) function cublasIcamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.3.2 icamin

If you use the cublas_v2 module, the interface for cublasIcamin is changed to the following:

```
integer(4) function cublasIcamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.3.3 scasum

If you use the cublas_v2 module, the interface for cublasScasum is changed to the following:

```
integer(4) function cublasScasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.7.3.4 caxpy

If you use the cublas_v2 module, the interface for cublasCaxpy is changed to the following:

```
integer(4) function cublasCaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.3.5 ccopy

If you use the cublas_v2 module, the interface for cublasCcopy is changed to the following:

```
integer(4) function cublasCcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.3.6 cdotc

If you use the cublas_v2 module, the interface for cublasCdotc is changed to the following:

```
integer(4) function cublasCdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

9.7.3.7 cdotu

If you use the cublas_v2 module, the interface for cublasCdotu is changed to the following:

```
integer(4) function cublasCdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(4), device :: res ! device or host variable
```

9.7.3.8 scnrm2

If you use the cublas_v2 module, the interface for cublasScnrm2 is changed to the following:

```
integer(4) function cublasScnrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x
  integer :: incx
  real(4), device :: res ! device or host variable
```

9.7.3.9 crot

If you use the cublas_v2 module, the interface for cublasCrot is changed to the following:

```
integer(4) function cublasCrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc ! device or host variable
  complex(4), device :: ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.3.10 csrot

If you use the cublas_v2 module, the interface for cublasCsrot is changed to the following:

```
integer(4) function cublasCsrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: sc, ss ! device or host variable
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.3.11 crotg

If you use the cublas_v2 module, the interface for cublasCrotg is changed to the following:

```
integer(4) function cublasCrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(4), device :: sa, sb, ss ! device or host variable
  real(4), device :: sc ! device or host variable
```

9.7.3.12 cscal

If you use the cublas_v2 module, the interface for cublasCscal is changed to the following:

```
integer(4) function cublasCscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

9.7.3.13 csscal

If you use the cublas_v2 module, the interface for cublasCsscal is changed to the following:

```
integer(4) function cublasCsscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(4), device :: a ! device or host variable
  complex(4), device, dimension(*) :: x
  integer :: incx
```

9.7.3.14 cswap

If you use the cublas_v2 module, the interface for cublasCswap is changed to the following:

```
integer(4) function cublasCswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(4), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.3.15 cgbmv

If you use the cublas_v2 module, the interface for cublasCgbmv is changed to the following:

```
integer(4) function cublasCgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.16 cgemv

If you use the cublas_v2 module, the interface for cublasCgemv is changed to the following:

```
integer(4) function cublasCgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.17 cgerc

If you use the cublas_v2 module, the interface for cublasCgerc is changed to the following:

```
integer(4) function cublasCgerc(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.7.3.18 cgeru

If you use the cublas_v2 module, the interface for cublasCgeru is changed to the following:

```
integer(4) function cublasCgeru(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.7.3.19 csymv

If you use the cublas_v2 module, the interface for cublasCsymv is changed to the following:

```
integer(4) function cublasCsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.20 csyr

If you use the cublas_v2 module, the interface for cublasCsyr is changed to the following:

```
integer(4) function cublasCsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
  complex(4), device :: alpha ! device or host variable
```

9.7.3.21 csyr2

If you use the cublas_v2 module, the interface for cublasCsyr2 is changed to the following:

```
integer(4) function cublasCsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha ! device or host variable
```

9.7.3.22 ctbmv

If you use the cublas_v2 module, the interface for cublasCtbmv is changed to the following:

```
integer(4) function cublasCtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.7.3.23 ctbsv

If you use the cublas_v2 module, the interface for cublasCtbsv is changed to the following:

```
integer(4) function cublasCtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.7.3.24 ctpmv

If you use the cublas_v2 module, the interface for cublasCtpmv is changed to the following:

```
integer(4) function cublasCtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

9.7.3.25 ctpsv

If you use the cublas_v2 module, the interface for cublasCtpsv is changed to the following:

```
integer(4) function cublasCtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
```

9.7.3.26 ctrmv

If you use the cublas_v2 module, the interface for cublasCtrmv is changed to the following:

```
integer(4) function cublasCtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.7.3.27 ctrsv

If you use the cublas_v2 module, the interface for cublasCtrsv is changed to the following:

```
integer(4) function cublasCtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x
```

9.7.3.28 chbmv

If you use the cublas_v2 module, the interface for cublasChbmv is changed to the following:

```
integer(4) function cublasChbmv(h, uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.29 chemv

If you use the cublas_v2 module, the interface for cublasChemv is changed to the following:

```
integer(4) function cublasChemv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.30 chpmv

If you use the cublas_v2 module, the interface for cublasChpmv is changed to the following:

```
integer(4) function cublasChpmv(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.31 cher

If you use the cublas_v2 module, the interface for cublasCher is changed to the following:

```
integer(4) function cublasCher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.7.3.32 cher2

If you use the cublas_v2 module, the interface for cublasCher2 is changed to the following:

```
integer(4) function cublasCher2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

9.7.3.33 chpr

If you use the cublas_v2 module, the interface for cublasChpr is changed to the following:

```
integer(4) function cublasChpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(4), device, dimension(*) :: a, x
  real(4), device :: alpha ! device or host variable
```

9.7.3.34 chpr2

If you use the cublas_v2 module, the interface for cublasChpr2 is changed to the following:

```
integer(4) function cublasChpr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(4), device, dimension(*) :: a, x, y
  complex(4), device :: alpha ! device or host variable
```

9.7.3.35 cgemm

If you use the cublas_v2 module, the interface for cublasCgemm is changed to the following:

```
integer(4) function cublasCgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.36 csymm

If you use the cublas_v2 module, the interface for cublasCsymm is changed to the following:

```
integer(4) function cublasCsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.37 csyrk

If you use the cublas_v2 module, the interface for cublasCsyrk is changed to the following:

```
integer(4) function cublasCsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.38 csyr2k

If you use the cublas_v2 module, the interface for cublasCsyr2k is changed to the following:

```
integer(4) function cublasCsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.39 csyrkx

If you use the cublas_v2 module, the interface for cublasCsyrkx is changed to the following:

```
integer(4) function cublasCsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.40 ctrmm

If you use the cublas_v2 module, the interface for cublasCtrmm is changed to the following:

```
integer(4) function cublasCtrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
```

9.7.3.41 ctrsm

If you use the cublas_v2 module, the interface for cublasCtrsm is changed to the following:

```
integer(4) function cublasCtrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(4), device, dimension(lda, *) :: a
```

(continues on next page)

(continued from previous page)

```
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

9.7.3.42 chemm

If you use the cublas_v2 module, the interface for cublasChemm is changed to the following:

```
integer(4) function cublasChemm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha, beta ! device or host variable
```

9.7.3.43 cherk

If you use the cublas_v2 module, the interface for cublasCherk is changed to the following:

```
integer(4) function cublasCherk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldc, *) :: c
  real(4), device :: alpha, beta ! device or host variable
```

9.7.3.44 cher2k

If you use the cublas_v2 module, the interface for cublasCher2k is changed to the following:

```
integer(4) function cublasCher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

9.7.3.45 cherkx

If you use the cublas_v2 module, the interface for cublasCherkx is changed to the following:

```
integer(4) function cublasCherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(4), device, dimension(lda, *) :: a
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device, dimension(ldc, *) :: c
  complex(4), device :: alpha ! device or host variable
  real(4), device :: beta ! device or host variable
```

9.7.4. Double Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the double precision complex BLAS and cuBLAS functions and subroutines.

9.7.4.1 izamax

If you use the cublas_v2 module, the interface for cublasIzamax is changed to the following:

```
integer(4) function cublasIzamax(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.4.2 izamin

If you use the cublas_v2 module, the interface for cublasIzamin is changed to the following:

```
integer(4) function cublasIzamin(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  integer, device :: res ! device or host variable
```

9.7.4.3 dzasum

If you use the cublas_v2 module, the interface for cublasDzasum is changed to the following:

```
integer(4) function cublasDzasum(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.7.4.4 zaxpy

If you use the cublas_v2 module, the interface for cublasZaxpy is changed to the following:

```
integer(4) function cublasZaxpy(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.4.5 zcopy

If you use the cublas_v2 module, the interface for cublasZcopy is changed to the following:

```
integer(4) function cublasZcopy(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.4.6 zdotc

If you use the cublas_v2 module, the interface for cublasZdotc is changed to the following:

```
integer(4) function cublasZdotc(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

9.7.4.7 zdotu

If you use the cublas_v2 module, the interface for cublasZdotu is changed to the following:

```
integer(4) function cublasZdotu(h, n, x, incx, y, incy, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
  complex(8), device :: res ! device or host variable
```

9.7.4.8 dznrm2

If you use the cublas_v2 module, the interface for cublasDznrm2 is changed to the following:

```
integer(4) function cublasDznrm2(h, n, x, incx, res)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

9.7.4.9 zrot

If you use the cublas_v2 module, the interface for cublasZrot is changed to the following:

```
integer(4) function cublasZrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc ! device or host variable
  complex(8), device :: ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.4.10 zsrot

If you use the cublas_v2 module, the interface for cublasZsrot is changed to the following:

```
integer(4) function cublasZsrot(h, n, x, incx, y, incy, sc, ss)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: sc, ss ! device or host variable
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.4.11 zrotg

If you use the cublas_v2 module, the interface for cublasZrotg is changed to the following:

```
integer(4) function cublasZrotg(h, sa, sb, sc, ss)
  type(cublasHandle) :: h
  complex(8), device :: sa, sb, ss ! device or host variable
  real(8), device :: sc ! device or host variable
```

9.7.4.12 zscal

If you use the cublas_v2 module, the interface for cublasZscal is changed to the following:

```
integer(4) function cublasZscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

9.7.4.13 zdscal

If you use the cublas_v2 module, the interface for cublasZdscal is changed to the following:

```
integer(4) function cublasZdscal(h, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: n
  real(8), device :: a ! device or host variable
  complex(8), device, dimension(*) :: x
  integer :: incx
```

9.7.4.14 zswap

If you use the cublas_v2 module, the interface for cublasZswap is changed to the following:

```
integer(4) function cublasZswap(h, n, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  complex(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

9.7.4.15 zgbmv

If you use the cublas_v2 module, the interface for cublasZgbmv is changed to the following:

```
integer(4) function cublasZgbmv(h, t, m, n, kl, ku, alpha, a, lda, x, incx, beta, y,
↪incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, kl, ku, lda, incx, incy
```

(continues on next page)

(continued from previous page)

```
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.16 zgemv

If you use the cublas_v2 module, the interface for cublasZgemv is changed to the following:

```
integer(4) function cublasZgemv(h, t, m, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: t
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.17 zgerc

If you use the cublas_v2 module, the interface for cublasZgerc is changed to the following:

```
integer(4) function cublasZgerc(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.7.4.18 zgeru

If you use the cublas_v2 module, the interface for cublasZgeru is changed to the following:

```
integer(4) function cublasZgeru(h, m, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: m, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.7.4.19 zsymv

If you use the cublas_v2 module, the interface for cublasZsymv is changed to the following:

```
integer(4) function cublasZsymv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.20 zsyr

If you use the cublas_v2 module, the interface for cublasZsyr is changed to the following:

```
integer(4) function cublasZsyr(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
  complex(8), device :: alpha ! device or host variable
```

9.7.4.21 zsyr2

If you use the cublas_v2 module, the interface for cublasZsyr2 is changed to the following:

```
integer(4) function cublasZsyr2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha ! device or host variable
```

9.7.4.22 ztbmv

If you use the cublas_v2 module, the interface for cublasZtbmv is changed to the following:

```
integer(4) function cublasZtbmv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.7.4.23 ztbsv

If you use the cublas_v2 module, the interface for cublasZtbsv is changed to the following:

```
integer(4) function cublasZtbsv(h, u, t, d, n, k, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, k, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.7.4.24 ztpmv

If you use the cublas_v2 module, the interface for cublasZtpmv is changed to the following:

```
integer(4) function cublasZtpmv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

9.7.4.25 ztpsv

If you use the cublas_v2 module, the interface for cublasZtpsv is changed to the following:

```
integer(4) function cublasZtpsv(h, u, t, d, n, a, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
```

9.7.4.26 ztrmv

If you use the cublas_v2 module, the interface for cublasZtrmv is changed to the following:

```
integer(4) function cublasZtrmv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.7.4.27 ztrsv

If you use the cublas_v2 module, the interface for cublasZtrsv is changed to the following:

```
integer(4) function cublasZtrsv(h, u, t, d, n, a, lda, x, incx)
  type(cublasHandle) :: h
  integer :: u, t, d
  integer :: n, incx, lda
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x
```

9.7.4.28 zhbmv

If you use the cublas_v2 module, the interface for cublasZhbmv is changed to the following:

```
integer(4) function cublasZhbmv(h, uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: k, n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.29 zhemv

If you use the cublas_v2 module, the interface for cublasZhemv is changed to the following:

```
integer(4) function cublasZhemv(h, uplo, n, alpha, a, lda, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, lda, incx, incy
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.30 zhpmv

If you use the cublas_v2 module, the interface for cublasZhpmv is changed to the following:

```
integer(4) function cublasZhpmv(h, uplo, n, alpha, a, x, incx, beta, y, incy)
  type(cublasHandle) :: h
  integer :: uplo
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.31 zher

If you use the cublas_v2 module, the interface for cublasZher is changed to the following:

```
integer(4) function cublasZher(h, t, n, alpha, x, incx, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, lda
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.7.4.32 zher2

If you use the cublas_v2 module, the interface for cublasZher2 is changed to the following:

```
integer(4) function cublasZher2(h, t, n, alpha, x, incx, y, incy, a, lda)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy, lda
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

9.7.4.33 zhpr

If you use the cublas_v2 module, the interface for cublasZhpr is changed to the following:

```
integer(4) function cublasZhpr(h, t, n, alpha, x, incx, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx
  complex(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable
```

9.7.4.34 zhpr2

If you use the cublas_v2 module, the interface for cublasZhpr2 is changed to the following:

```
integer(4) function cublasZhpr2(h, t, n, alpha, x, incx, y, incy, a)
  type(cublasHandle) :: h
  integer :: t
  integer :: n, incx, incy
  complex(8), device, dimension(*) :: a, x, y
  complex(8), device :: alpha ! device or host variable
```

9.7.4.35 zgemm

If you use the cublas_v2 module, the interface for cublasZgemm is changed to the following:

```
integer(4) function cublasZgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasHandle) :: h
  integer :: transa, transb
  integer :: m, n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.36 zsymm

If you use the cublas_v2 module, the interface for cublasZsymm is changed to the following:

```
integer(4) function cublasZsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.37 zsyrk

If you use the cublas_v2 module, the interface for cublasZsyrk is changed to the following:

```
integer(4) function cublasZsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.38 zsyr2k

If you use the cublas_v2 module, the interface for cublasZsyr2k is changed to the following:

```
integer(4) function cublasZsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.39 zsyrkx

If you use the cublas_v2 module, the interface for cublasZsyrkx is changed to the following:

```
integer(4) function cublasZsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.40 ztrmm

If you use the cublas_v2 module, the interface for cublasZtrmm is changed to the following:

```
integer(4) function cublasZtrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb, c, ldc)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
```

9.7.4.41 ztrsm

If you use the cublas_v2 module, the interface for cublasZtrsm is changed to the following:

```
integer(4) function cublasZtrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ldb)
  type(cublasHandle) :: h
  integer :: side, uplo, transa, diag
  integer :: m, n, lda, ldb
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device :: alpha ! device or host variable
```

9.7.4.42 zhemm

If you use the cublas_v2 module, the interface for cublasZhemm is changed to the following:

```
integer(4) function cublasZhemm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ldc)
  type(cublasHandle) :: h
  integer :: side, uplo
  integer :: m, n, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha, beta ! device or host variable
```

9.7.4.43 zherk

If you use the cublas_v2 module, the interface for cublasZherk is changed to the following:

```
integer(4) function cublasZherk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldc, *) :: c
  real(8), device :: alpha, beta ! device or host variable
```

9.7.4.44 zher2k

If you use the cublas_v2 module, the interface for cublasZher2k is changed to the following:

```
integer(4) function cublasZher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

9.7.4.45 zherkx

If you use the cublas_v2 module, the interface for cublasZherkx is changed to the following:

```
integer(4) function cublasZherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasHandle) :: h
  integer :: uplo, trans
  integer :: n, k, lda, ldb, ldc
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(ldb, *) :: b
  complex(8), device, dimension(ldc, *) :: c
  complex(8), device :: alpha ! device or host variable
  real(8), device :: beta ! device or host variable
```

9.8. CUBLAS XT Module Functions

This section contains interfaces to the cuBLAS XT Module Functions. Users can access this module by inserting the line `use cublasXt` into the program unit. The cublasXt library is a host-side library, which supports multiple GPUs. Here is an example:

```
subroutine testxt(n)
use cublasXt
complex*16 :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = cmplx(1.0d0,0.0d0)
b = cmplx(2.0d0,0.0d0)
c = cmplx(-1.0d0,0.0d0)
alpha = cmplx(1.0d0,0.0d0)
beta = cmplx(0.0d0,0.0d0)
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtZgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
```

(continues on next page)

(continued from previous page)

```
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(dble(c).eq.2.0d0*n)) then
    print *,"Test PASSED"
else
    print *,"Test FAILED"
endif
end
```

The `cublasXt` module contains all the types and definitions from the `cublas` module, and these additional types and enumerations:

```
TYPE cublasXtHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

```
! Pinned memory mode
enum, bind(c)
    enumerator :: CUBLASXT_PINNING_DISABLED=0
    enumerator :: CUBLASXT_PINNING_ENABLED=1
end enum
```

```
! cublasXtOpType
enum, bind(c)
    enumerator :: CUBLASXT_FLOAT=0
    enumerator :: CUBLASXT_DOUBLE=1
    enumerator :: CUBLASXT_COMPLEX=2
    enumerator :: CUBLASXT_DOUBLECOMPLEX=3
end enum
```

```
! cublasXtBlasOp
enum, bind(c)
    enumerator :: CUBLASXT_GEMM=0
    enumerator :: CUBLASXT_SYRK=1
    enumerator :: CUBLASXT_HERK=2
    enumerator :: CUBLASXT_SYMM=3
    enumerator :: CUBLASXT_HEMM=4
    enumerator :: CUBLASXT_TRSM=5
    enumerator :: CUBLASXT_SYR2K=6
    enumerator :: CUBLASXT_HER2K=7
    enumerator :: CUBLASXT_SPMM=8
    enumerator :: CUBLASXT_SYRKX=9
    enumerator :: CUBLASXT_HERKX=10
    enumerator :: CUBLASXT_TRMM=11
    enumerator :: CUBLASXT_ROUTINE_MAX=12
end enum
```

9.8.1. cublasXtCreate

This function initializes the cublasXt API and creates a handle to an opaque structure holding the cublasXT library context. It allocates hardware resources on the host and device and must be called prior to making any other cublasXt API library calls.

```
integer(4) function cublasXtcreate(h)
  type(cublasXtHandle) :: h
```

9.8.2. cublasXtDestroy

This function releases hardware resources used by the cublasXt API context. This function is usually the last call with a particular handle to the cublasXt API.

```
integer(4) function cublasXtdestroy(h)
  type(cublasXtHandle) :: h
```

9.8.3. cublasXtDeviceSelect

This function allows the user to provide the number of GPU devices and their respective Ids that will participate to the subsequent cublasXt API math function calls. This function will create a cuBLAS context for every GPU provided in that list. Currently the device configuration is static and cannot be changed between math function calls. In that regard, this function should be called only once after cublasXtCreate. To be able to run multiple configurations, multiple cublasXt API contexts should be created.

```
integer(4) function cublasXtdeviceselect(h, ndevices, deviceid)
  type(cublasXtHandle) :: h
  integer :: ndevices
  integer, dimension(*) :: deviceid
```

9.8.4. cublasXtSetBlockDim

This function allows the user to set the block dimension used for the tiling of the matrices for the subsequent Math function calls. Matrices are split in square tiles of blockDim x blockDim dimension. This function can be called anytime and will take effect for the following math function calls. The block dimension should be chosen in a way to optimize the math operation and to make sure that the PCI transfers are well overlapped with the computation.

```
integer(4) function cublasXtsetblockdim(h, blockdim)
  type(cublasXtHandle) :: h
  integer :: blockdim
```

9.8.5. cublasXtGetBlockDim

This function allows the user to query the block dimension used for the tiling of the matrices.

```
integer(4) function cublasXtgetblockdim(h, blockdim)
  type(cublasXtHandle) :: h
  integer :: blockdim
```

9.8.6. cublasXtSetCpuRoutine

This function allows the user to provide a CPU implementation of the corresponding BLAS routine. This function can be used with the function cublasXtSetCpuRatio() to define an hybrid computation between the CPU and the GPUs. Currently the hybrid feature is only supported for the xGEMM routines.

```
integer(4) function cublasXtsetcpuroutine(h, blasop, blastype)
  type(cublasXtHandle) :: h
  integer :: blasop, blastype
```

9.8.7. cublasXtSetCpuRatio

This function allows the user to define the percentage of workload that should be done on a CPU in the context of an hybrid computation. This function can be used with the function cublasXtSetCpuRoutine() to define an hybrid computation between the CPU and the GPUs. Currently the hybrid feature is only supported for the xGEMM routines.

```
integer(4) function cublasXtsetcpuratio(h, blasop, blastype, ratio)
  type(cublasXtHandle) :: h
  integer :: blasop, blastype
  real(4) :: ratio
```

9.8.8. cublasXtSetPinningMemMode

This function allows the user to enable or disable the Pinning Memory mode. When enabled, the matrices passed in subsequent cublasXt API calls will be pinned/unpinned using the CUDART routine cudaHostRegister and cudaHostUnregister respectively if the matrices are not already pinned. If a matrix happened to be pinned partially, it will also not be pinned. Pinning the memory improve PCI transfer performace and allows to overlap PCI memory transfer with computation. However pinning/unpinning the memory takes some time which might not be amortized. It is advised that the user pins the memory on its own using cudaMallocHost or cudaHostRegister and unpins it when the computation sequence is completed. By default, the Pinning Memory mode is disabled.

```
integer(4) function cublasXtsetpinningmemmode(h, mode)
  type(cublasXtHandle) :: h
  integer :: mode
```

9.8.9. cublasXtGetPinningMemMode

This function allows the user to query the Pinning Memory mode. By default, the Pinning Memory mode is disabled.

```
integer(4) function cublasXtgetpinningmemmode(h, mode)
  type(cublasXtHandle) :: h
  integer :: mode
```

9.8.10. cublasXtSgemm

SGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtsgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.11. cublasXtSsymm

SSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtssymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.12. cublasXtSsyrk

SSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtssyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.13. cublasXtSsyr2k

SSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtssyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.14. cublasXtSsyrkx

SSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtssyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.15. cublasXtStrmm

STRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
integer(4) function cublasXtstrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha
```

9.8.16. cublasXtStrsm

STRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasXtstrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldb, *) :: b
  real(4) :: alpha
```

9.8.17. cublasXtSspmm

SSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtsspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta, c,
↪ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  real(4), dimension(*) :: ap
  real(4), dimension(ldb, *) :: b
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.18. cublasXtCgemm

CGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtcgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.19. cublasXtChemm

CHEMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

```
integer(4) function cublasXtchemm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.20. cublasXtCherk

CHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtcherk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

9.8.21. cublasXtCher2k

CHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtcher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
  real(4) :: beta
```

9.8.22. cublasXtCherkx

CHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtcherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
  real(4) :: beta
```

9.8.23. cublasXtCsymm

CSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtcsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.24. cublasXtCsyrk

CSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtcsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.25. cublasXtCsyr2k

CSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtcsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.26. cublasXtCsyrkx

CSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtcsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.27. cublasXtCtrmm

CTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
integer(4) function cublasXtctrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha
```

9.8.28. cublasXtCtrsm

CTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasXtctrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  complex(4), dimension(lda, *) :: a
  complex(4), dimension(ldb, *) :: b
  complex(4) :: alpha
```

9.8.29. cublasXtCspmm

CSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtcspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta, c,
↪ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  complex(4), dimension(*) :: ap
  complex(4), dimension(ldb, *) :: b
  complex(4), dimension(ldc, *) :: c
  complex(4) :: alpha, beta
```

9.8.30. cublasXtDgemm

DGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtdgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.31. cublasXtDsymm

DSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtdsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.32. cublasXtDsyrk

DSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtdsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.33. cublasXtDsyr2k

DSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtdsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.34. cublasXtDsyrkx

DSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtdsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.35. cublasXtDtrmm

DTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A), where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T.

```
integer(4) function cublasXtdtrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha
```

9.8.36. cublasXtDtrsm

DTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T. The matrix X is overwritten on B.

```
integer(4) function cublasXtdtrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  real(8), dimension(lda, *) :: a
  real(8), dimension(ldb, *) :: b
  real(8) :: alpha
```

9.8.37. cublasXtDspmm

DSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtdspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta, c,
↪ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  real(8), dimension(*) :: ap
  real(8), dimension(ldb, *) :: b
  real(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.38. cublasXtZgemm

ZGEMM performs one of the matrix-matrix operations C := alpha*op(A)*op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T or op(X) = X**H, alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
integer(4) function cublasXtzgemm(h, transa, transb, m, n, k, alpha, a, lda, b, ldb,
↪beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: transa, transb
  integer(kind=c_intptr_t) :: m, n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.39. cublasXtZhemm

ZHEMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is an hermitian matrix and B and C are m by n matrices.

```
integer(4) function cublasXtzhemm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.40. cublasXtZherk

ZHERK performs one of the hermitian rank k operations C := alpha*A*A**H + beta*C, or C := alpha*A**H*A + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtzherk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

9.8.41. cublasXtZher2k

ZHER2K performs one of the hermitian rank 2k operations C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C, where alpha and beta are scalars with beta real, C is an n by n hermitian matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtzher2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
  real(8) :: beta
```

9.8.42. cublasXtZherkx

ZHERKX performs a variation of the hermitian rank k operations C := alpha*A*B**H + beta*C, where alpha and beta are real scalars, C is an n by n hermitian matrix stored in lower or upper mode, and A and B are n by k matrices. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtzherkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
  real(8) :: beta
```

9.8.43. cublasXtZsymm

ZSYMM performs one of the matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a symmetric matrix and B and C are m by n matrices.

```
integer(4) function cublasXtzsymm(h, side, uplo, m, n, alpha, a, lda, b, ldb, beta, c,
↪ ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.44. cublasXtZsyrk

ZSYRK performs one of the symmetric rank k operations C := alpha*A*A**T + beta*C, or C := alpha*A**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k matrix in the first case and a k by n matrix in the second case.

```
integer(4) function cublasXtzsyrk(h, uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.45. cublasXtZsyr2k

ZSYR2K performs one of the symmetric rank 2k operations C := alpha*A*B**T + alpha*B*A**T + beta*C, or C := alpha*A**T*B + alpha*B**T*A + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.

```
integer(4) function cublasXtzsyr2k(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.46. cublasXtZsyrkx

ZSYRKX performs a variation of the symmetric rank k update C := alpha*A*B**T + beta*C, where alpha and beta are scalars, C is an n by n symmetric matrix stored in lower or upper mode, and A and B are n by k matrices. This routine can be used when B is in such a way that the result is guaranteed to be symmetric. See the CUBLAS documentation for more details.

```
integer(4) function cublasXtzsyrkx(h, uplo, trans, n, k, alpha, a, lda, b, ldb, beta,
↪c, ldc)
  type(cublasXtHandle) :: h
  integer :: uplo, trans
  integer(kind=c_intptr_t) :: n, k, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.8.47. cublasXtZtrmm

ZTRMM performs one of the matrix-matrix operations B := alpha*op(A)*B, or B := alpha*B*op(A) where alpha is a scalar, B is an m by n matrix, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H.

```
integer(4) function cublasXtztrmm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb, c, ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha
```

9.8.48. cublasXtZtrsm

ZTRSM solves one of the matrix equations op(A)*X = alpha*B, or X*op(A) = alpha*B, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and op(A) is one of op(A) = A or op(A) = A**T or op(A) = A**H. The matrix X is overwritten on B.

```
integer(4) function cublasXtztrsm(h, side, uplo, transa, diag, m, n, alpha, a, lda, b,
↪ ldb)
  type(cublasXtHandle) :: h
  integer :: side, uplo, transa, diag
  integer(kind=c_intptr_t) :: m, n, lda, ldb
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldb, *) :: b
  complex(8) :: alpha
```

9.8.49. cublasXtZspmm

ZSPMM performs one of the symmetric packed matrix-matrix operations C := alpha*A*B + beta*C, or C := alpha*B*A + beta*C, where alpha and beta are scalars, A is a n by n symmetric matrix stored in packed format, and B and C are m by n matrices.

```
integer(4) function cublasXtzspmm(h, side, uplo, m, n, alpha, ap, b, ldb, beta, c,
↪ldc)
  type(cublasXtHandle) :: h
  integer :: side, uplo
  integer(kind=c_intptr_t) :: m, n, ldb, ldc
  complex(8), dimension(*) :: ap
  complex(8), dimension(ldb, *) :: b
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

9.9. CUBLAS MP Module Functions

This section contains interfaces to the cuBLAS MP Module Functions. Users can access this module by inserting the line `use cublasMp` into the program unit. The cublasMp library is a host-side library which operates on distributed device data, and which supports multiple processes and GPUs. It is based on the ScaLAPACK PBLAS library.

Beginning with the 26.3 release, cublasMp version 0.8.0 renames a few APIs. See cuBLASMp v0.8.0 for details.

Beginning with the 25.9 release, cublasMp version 0.5.0 has transitioned from using the Communication Abstraction Library (CAL, `cal_comm`) to using NCCL directly (`ncclComm`). This is a breaking change in the communicator type passed to functions such as `cublasMpGridCreate`, `cublasMpGemr2D`, and `cublasMpTrmr2D`. Users must now initialize NCCL communicators and pass `type(ncclComm)` instead of `type(cal_comm)`. The `nccl` module, included via `use cublasMp`, provides the necessary type definitions. For migration guidance, see the cuBLASMp release notes and the Migrating from CAL to NCCL documentation.

Beginning with the 25.1 release, the cublasMp library has a newer API for CUDA versions > 12.0, specifically cublasMp version 0.3.0 and higher. For users of CUDA versions <= 11.8, the old module has been

renamed, and you can access it by inserting the line `use cublasMp02` in the program unit. One major difference in version 0.3.x is that all cublasMp functions now return a `type(cublasMpStatus)` rather than an integer(4). There are other additions and changes which we will point out in the individual descriptions below. For complete documentation of the Fortran interfaces for cublasMp 0.2.x, please see the documentation from a 2024 NVHPC release.

Some overloaded operations for comparing and assigning `type(cublasMpStatus)` variables and expressions are provided in the new module.

The `cublasMp` module contains all the common types and definitions from the `cublas` module, types and interfaces from the `nccl` module, and these additional types and enumerations:

```
! Version information
integer, parameter :: CUBLASMP_VER_MAJOR = 0
integer, parameter :: CUBLASMP_VER_MINOR = 3
integer, parameter :: CUBLASMP_VER_PATCH = 0
integer, parameter :: CUBLASMP_VERSION = &
     (CUBLASMP_VER_MAJOR * 1000 + CUBLASMP_VER_MINOR * 100 + CUBLASMP_VER_PATCH)
```

```
! New status type, with version 0.3.0
TYPE cublasMpStatus
  integer(4) :: stat
END TYPE
TYPE(cublasMpStatus), parameter :: &
  CUBLASMP_STATUS_SUCCESS                = cublasMpStatus(0), &
  CUBLASMP_STATUS_NOT_INITIALIZED        = cublasMpStatus(1), &
  CUBLASMP_STATUS_ALLOCATION_FAILED      = cublasMpStatus(2), &
  CUBLASMP_STATUS_INVALID_VALUE          = cublasMpStatus(3), &
  CUBLASMP_STATUS_ARCHITECTURE_MISMATCH  = cublasMpStatus(4), &
  CUBLASMP_STATUS_EXECUTION_FAILED       = cublasMpStatus(5), &
  CUBLASMP_STATUS_INTERNAL_ERROR         = cublasMpStatus(6), &
  CUBLASMP_STATUS_NOT_SUPPORTED          = cublasMpStatus(7)
```

```
! Grid Layout
TYPE cublasMpGridLayout
  integer(4) :: grid
END TYPE
TYPE(cublasMpGridLayout), parameter :: &
  CUBLASMP_GRID_LAYOUT_COL_MAJOR = cublasMpGridLayout(0), &
  CUBLASMP_GRID_LAYOUT_ROW_MAJOR = cublasMpGridLayout(1)
```

```
! Matmul Descriptor Attributes
TYPE cublasMpMatmulDescriptorAttribute
  integer(4) :: attr
END TYPE
TYPE(cublasMpMatmulDescriptorAttribute), parameter :: &
  CUBLASMP_MATMUL_DESCRIPTOR_ATTRIBUTE_TRANSA       =
↪cublasMpMatmulDescriptorAttribute(0), &
  CUBLASMP_MATMUL_DESCRIPTOR_ATTRIBUTE_TRANSB       =
↪cublasMpMatmulDescriptorAttribute(1), &
  CUBLASMP_MATMUL_DESCRIPTOR_ATTRIBUTE_COMPUTE_TYPE =
↪cublasMpMatmulDescriptorAttribute(2), &
  CUBLASMP_MATMUL_DESCRIPTOR_ATTRIBUTE_ALGO_TYPE    =
↪cublasMpMatmulDescriptorAttribute(3)
```

```
! Matmul Algorithm Type
TYPE cublasMpMatmulAlgoType
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: atyp
END TYPE
TYPE(cublasMpMatmulAlgoType), parameter :: &
  CUBLASMP_MATMUL_ALGO_TYPE_DEFAULT          = cublasMpMatmulAlgoType(0), &
  CUBLASMP_MATMUL_ALGO_TYPE_SPLIT_P2P        = cublasMpMatmulAlgoType(1), &
  CUBLASMP_MATMUL_ALGO_TYPE_SPLIT_MULTICAST  = cublasMpMatmulAlgoType(2), &
  CUBLASMP_MATMUL_ALGO_TYPE_ATOMIC_P2P       = cublasMpMatmulAlgoType(3), &
  CUBLASMP_MATMUL_ALGO_TYPE_ATOMIC_MULTICAST = cublasMpMatmulAlgoType(4)
```

```
! Emulation Strategy
TYPE cublasMpEmulationStrategy
  integer(4) :: strategy
END TYPE
TYPE(cublasMpEmulationStrategy), parameter :: &
  CUBLASMP_EMULATION_STRATEGY_DEFAULT    = cublasMpEmulationStrategy(0), &
  CUBLASMP_EMULATION_STRATEGY_PERFORMANT = cublasMpEmulationStrategy(1), &
  CUBLASMP_EMULATION_STRATEGY_EAGER      = cublasMpEmulationStrategy(2)
```

```
TYPE cublasMpHandle
  TYPE(C_PTR)  :: handle
END TYPE
```

```
TYPE cublasMpGrid
  TYPE(C_PTR)  :: handle
END TYPE
```

```
TYPE cublasMpMatrixDescriptor
  TYPE(C_PTR)  :: handle
END TYPE
```

```
TYPE cublasMpMatmulDescriptor
  TYPE(C_PTR)  :: handle
END TYPE
```

9.9.1. cublasMpCreate

This function initializes the cublasMp API and creates a handle to an opaque structure holding the cublasMp library context. It allocates hardware resources on the host and device and must be called prior to making any other cublasMp library calls.

```
type(cublasMpStatus) function cublasMpCreate(handle, stream)
  type(cublasMpHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

9.9.2. cublasMpDestroy

This function releases resources used by the cublasMp handle and context.

```
type(cublasMpStatus) function cublasMpDestroy(handle)
  type(cublasMpHandle) :: handle
```

9.9.3. cublasMpSetStream

This function sets the CUDA stream to be used in the cublasMp computations.

```
type(cublasMpStatus) function cublasMpSetStream(handle, stream)
  type(cublasMpHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

9.9.4. cublasMpGetStream

This function returns the current CUDA stream used in the cublasMp computations.

```
type(cublasMpStatus) function cublasMpGetStream(handle, stream)
  type(cublasMpHandle) :: handle
  integer(kind=cuda_stream_kind()) :: stream
```

9.9.5. cublasMpSetEmulationStrategy

This function sets the emulation strategy for cublasMp computations.

```
type(cublasMpStatus) function cublasMpSetEmulationStrategy(handle, emulationStrategy)
  type(cublasMpHandle) :: handle
  type(cublasMpEmulationStrategy) :: emulationStrategy
```

9.9.6. cublasMpGetEmulationStrategy

This function retrieves the emulation strategy used in cublasMp computations.

```
type(cublasMpStatus) function cublasMpGetEmulationStrategy(handle, emulationStrategy)
  type(cublasMpHandle) :: handle
  type(cublasMpEmulationStrategy) :: emulationStrategy
```

9.9.7. cublasMpSetFixedPointEmulationMaxMantissaBitCount

This function sets the maximum number of mantissa bits used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function
↪cublasMpSetFixedPointEmulationMaxMantissaBitCount(handle, bitCount)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: bitCount
```

9.9.8. cublasMpGetFixedPointEmulationMaxMantissaBitCount

This function returns the maximum number of mantissa bits used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function
↪cublasMpGetFixedPointEmulationMaxMantissaBitCount(handle, bitCount)
   type(cublasMpHandle) :: handle
   integer(4), intent(out) :: bitCount
```

9.9.9. cublasMpSetFixedPointEmulationMantissaBitOffset

This function sets the mantissa bit offset used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function cublasMpSetFixedPointEmulationMantissaBitOffset(handle,
↪bitOffset)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: bitOffset
```

9.9.10. cublasMpGetFixedPointEmulationMantissaBitOffset

This function returns the mantissa bit offset used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function cublasMpGetFixedPointEmulationMantissaBitOffset(handle,
↪bitOffset)
   type(cublasMpHandle) :: handle
   integer(4), intent(out) :: bitOffset
```

9.9.11. cublasMpSetFixedPointEmulationMantissaBitCountPointer

This function sets a device pointer supplying the mantissa bit count used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function
↪cublasMpSetFixedPointEmulationMantissaBitCountPointer(handle, ptr)
   type(cublasMpHandle) :: handle
   type(c_ptr), intent(in) :: ptr
```

9.9.12. cublasMpGetFixedPointEmulationMantissaBitCountPointer

This function returns a device pointer supplying the mantissa bit count used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function
↪cublasMpGetFixedPointEmulationMantissaBitCountPointer(handle, ptr)
   type(cublasMpHandle) :: handle
   type(c_ptr), intent(out) :: ptr
```

9.9.13. cublasMpSetFixedPointEmulationMantissaControl

This function sets the mantissa control mode used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function cublasMpSetFixedPointEmulationMantissaControl(handle,
↪mantissaControl)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: mantissaControl
```

9.9.14. cublasMpGetFixedPointEmulationMantissaControl

This function returns the mantissa control mode used by fixed-point emulation. It returns CUBLASMP_STATUS_NOT_SUPPORTED on hardware or library versions where fixed-point emulation is unavailable.

```
type(cublasMpStatus) function cublasMpGetFixedPointEmulationMantissaControl(handle,
↪mantissaControl)
   type(cublasMpHandle) :: handle
   integer(4), intent(out) :: mantissaControl
```

9.9.15. cublasMpSetEmulationSpecialValuesSupport

This function sets the mask controlling which special floating-point values are supported when emulation is in use. It returns `CUBLASMP_STATUS_NOT_SUPPORTED` on hardware or library versions where emulation is unavailable.

```
type(cublasMpStatus) function cublasMpSetEmulationSpecialValuesSupport(handle, mask)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: mask
```

9.9.16. cublasMpGetEmulationSpecialValuesSupport

This function returns the mask controlling which special floating-point values are supported when emulation is in use. It returns `CUBLASMP_STATUS_NOT_SUPPORTED` on hardware or library versions where emulation is unavailable.

```
type(cublasMpStatus) function cublasMpGetEmulationSpecialValuesSupport(handle, mask)
   type(cublasMpHandle) :: handle
   integer(4), intent(out) :: mask
```

9.9.17. cublasMpGetVersion

This function returns the version number of the cublasMp library.

```
type(cublasMpStatus) function cublasMpGetVersion(version)
  integer(4) :: version
```

9.9.18. cublasMpGetStatusString

This function returns a C pointer to a NUL-terminated string describing the given status value. Use `c_f_pointer` from `iso_c_binding` to access the characters.

```
type(c_ptr) function cublasMpGetStatusString(status)
   type(cublasMpStatus), intent(in) :: status
```

9.9.19. cublasMpGridCreate

This function initializes the grid data structure used in the cublasMp library. It takes a communicator, and other information related to the data layout as inputs. Starting in version 0.3.0, it no longer takes a handle argument.

```
type(cublasMpStatus) function cublasMpGridCreate(nprow, npcol, &
          layout, comm, grid)
  integer(8) :: nprow, npcol
```

(continues on next page)

(continued from previous page)

```
  type(cublasMpGridLayout) :: layout ! usually column major in Fortran
  type(ncclComm) :: comm
  type(cublasMpGrid), intent(out) :: grid
```

9.9.20. cublasMpGridDestroy

This function releases the grid data structure used in the cublasMp library. Starting in version 0.3.0, it no longer takes a handle argument.

```
type(cublasMpStatus) function cublasMpGridDestroy(grid)
  type(cublasMpGrid) :: grid
```

9.9.21. cublasMpBufferRegister

This function registers a device buffer with the cublasMp grid for use in multi-GPU operations.

```
type(cublasMpStatus) function cublasMpBufferRegister(grid, ptr, size)
  type(cublasMpGrid) :: grid
  type(C_PTR) :: ptr
  integer(8) :: size
```

9.9.22. cublasMpBufferDeregister

This function deregisters a device buffer previously registered with the cublasMp grid.

```
type(cublasMpStatus) function cublasMpBufferDeregister(grid, ptr)
  type(cublasMpGrid) :: grid
  type(C_PTR) :: ptr
```

9.9.23. cublasMpMalloc

This function allocates memory registered with the cublasMp grid.

```
type(cublasMpStatus) function cublasMpMalloc(grid, ptr, size)
  type(cublasMpGrid) :: grid
  type(C_PTR) :: ptr
  integer(8) :: size
```

9.9.24. cublasMpFree

This function frees memory previously allocated with cublasMpMalloc.

```
type(cublasMpStatus) function cublasMpFree(grid, ptr)
  type(cublasMpGrid) :: grid
  type(C_PTR) :: ptr
```

9.9.25. cublasMpMatrixDescriptorCreate

This function initializes the matrix descriptor object used in the cublasMp library. It takes the number of rows (M) and the number of columns (N) in the global array, along with the blocking factor over each dimension. RSRC and CSRC must currently be 0. LLD is the leading dimension of the local matrix, after blocking and distributing the matrix. Starting in version 0.3.0, it no longer takes a handle argument.

```
type(cublasMpStatus) function cublasMpMatrixDescriptorCreate(M, N, MB, NB, &
          RSRC, CSRC, LLD, dataType, grid, descr)
  integer(8) :: M, N, MB, NB, RSRC, CSRC, LLD
  type(cudaDataType) :: dataType
  type(cublasMpGrid) :: grid
  type(cublasMpMatrixDescriptor), intent(out)  :: descr
```

9.9.26. cublasMpMatrixDescriptorDestroy

This function frees the matrix descriptor object used in the cublasMp library. Starting in version 0.3.0, it no longer takes a handle argument.

```
type(cublasMpStatus) function cublasMpMatrixDescriptorDestroy(descr)
  type(cublasMpMatrixDescriptor) :: descr
```

9.9.27. cublasMpMatrixDescriptorInit

This function initializes the values within the matrix descriptor object used in the cublasMp library. It takes the number of rows (M) and the number of columns (N) in the global array, along with the blocking factor over each dimension. RSRC and CSRC must currently be 0. LLD is the leading dimension of the local matrix, after blocking and distributing the matrix.

```
type(cublasMpStatus) function cublasMpMatrixDescriptorInit(M, N, MB, NB, &
          RSRC, CSRC, LLD, dataType, grid, descr)
  integer(8) :: M, N, MB, NB, RSRC, CSRC, LLD
  type(cudaDataType) :: dataType
  type(cublasMpGrid) :: grid
  type(cublasMpMatrixDescriptor), intent(out)  :: descr
```

9.9.28. cublasMpNumroc

This function computes (and returns) the local number of rows or columns of a distributed matrix, similar to the ScaLAPACK NUMROC function.

```
type(cublasMpStatus) function cublasMpNumroc(N, NB, iproc, isrcproc, nprocs)
  integer(8) :: N, NB
  integer(4) :: iproc, isrcproc, nprocs
```

9.9.29. cublasMpMatmulDescriptorCreate

This function initializes the matmul descriptor object used in the cublasMp library.

```
type(cublasMpStatus) function cublasMpMatmulDescriptorCreate(descr, computeType)
  type(cublasMpMatmulDescriptor)  :: descr
  type(cublasComputeType) :: computeType
```

9.9.30. cublasMpMatmulDescriptorDestroy

This function destroys the matmul descriptor object used in the cublasMp library.

```
type(cublasMpStatus) function cublasMpMatmulDescriptorDestroy(descr)
  type(cublasMpMatmulDescriptor)  :: descr
```

9.9.31. cublasMpMatmulDescriptorInit

This function re-initializes an existing matmul descriptor with the given compute type, without allocating a new descriptor.

```
type(cublasMpStatus) function cublasMpMatmulDescriptorInit(descr, computeType)
   type(cublasMpMatmulDescriptor) :: descr
   type(cublasComputeType) :: computeType
```

9.9.32. cublasMpMatmulDescriptorSetAttribute

This function sets attributes within the matmul descriptor object used in the cublasMp library.

```
type(cublasMpStatus) function cublasMpMatmulDescriptorSetAttribute(descr, attr, &
          buf, sizeInBytes)
  type(cublasMpMatmulDescriptor)  :: descr
  type(cublasMpMatmulDescriptorAttribute)  :: attr
  integer(1) :: buf(sizeInBytes) ! Any type, kind, or rank allowed
  integer(8) :: sizeInBytes
```

9.9.33. cublasMpMatmulDescriptorGetAttribute

This function retrieves attributes within the matmul descriptor object used in the cublasMp library.

```
type(cublasMpStatus) function cublasMpMatmulDescriptorGetAttribute(descr, attr, &
          buf, sizeInBytes, sizeWritten)
  type(cublasMpMatmulDescriptor)  :: descr
  type(cublasMpMatmulDescriptorAttribute)  :: attr
  integer(1) :: buf(sizeInBytes) ! Any type, kind, or rank allowed
  integer(8) :: sizeInBytes, sizeWritten
```

9.9.34. cublasMpGemr2D_bufferSize

This functions computes the workspace requirements of cublasMpGemr2D

```
type(cublasMpStatus) function cublasMpGemr2D_bufferSize(handle, M, N, &
   A, IA, JA, descrA, B, IB, JB, descrB, &
   devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes, comm)
   type(cublasMpHandle) :: handle
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   type(ncclComm) :: comm
```

9.9.35. cublasMpGemr2D

This functions copies a matrix from one distributed form to another. The layout of each matrix is defined in the matrix descriptor. M and N are the global matrix dimensions. IA, JA, IB, and JB are 1-based, and typically equal to 1 for a full matrix.

```
type(cublasMpStatus) function cublasMpGemr2D(handle, M, N,  &
   A, IA, JA, descrA, B, IB, JB, descrB, &
   bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes, comm)
   type(cublasMpHandle) :: handle
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
   type(ncclComm) :: comm
```

9.9.36. cublasMpTrmr2D_bufferSize

This functions computes the workspace requirements of cublasMpTrmr2D

```
type(cublasMpStatus) function cublasMpTrmr2D_bufferSize(handle, uplo, diag, &
   M, N, A, IA, JA, descrA, B, IB, JB, descrB, &
   devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes, comm)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: uplo, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   type(ncclComm) :: comm
```

9.9.37. cublasMpTrmr2D

This functions copies a trapezoidal matrix from one distributed form to another. The layout of each matrix is defined in the matrix descriptor. M and N are the global matrix dimensions. IA, JA, IB, and JB are 1-based, and typically equal to 1 for a full matrix.

```
type(cublasMpStatus) function cublasMpTrmr2D(handle, uplo, diag,  &
   M, N, A, IA, JA, descrA, B, IB, JB, descrB, &
   bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes, comm)
   type(cublasMpHandle) :: handle
   integer(4), intent(in) :: uplo, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
   type(ncclComm) :: comm
```

9.9.38. cublasMpGemm_bufferSize

This functions computes the workspace requirements of cublasMpGemm.

```
type(cublasMpStatus) function cublasMpGemm_bufferSize(handle, transA, transB, M, N, K,
↪ &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: transA, transB
   integer(8), intent(in) :: M, N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.39. cublasMpGemm

This is the multi-processor version of the BLAS GEMM operation, similar to the ScaLAPACK PBLAS functions pdgemm, pzgemm, etc.

GEMM performs one of the matrix-matrix operations

C := alpha*op(A)*op(B) + beta*C,

where op(X) is one of

op(X) = X or op(X) = X**T,

alpha and beta are scalars, and A, B, and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix. The data for A, B, and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB, and descrC via the `cublasMpMatrixDescriptorCreate()` function. The datatype is also specified then. M, N, and K are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpGemm(handle, transA, transB, M, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: transA, transB
   integer(8), intent(in) :: M, N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceInBytes)  ! Any type
```

9.9.40. cublasMpMatmul_bufferSize

This functions computes the workspace requirements of cublasMpMatmul.

```
type(cublasMpStatus) function cublasMpMatmul_bufferSize(handle, matmulDescr, M, N, K,
↪&
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   D, ID, JD, descrD, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   type(cublasMpMatmulDescriptor) :: matmulDescr
   integer(8), intent(in) :: M, N, K, IA, JA, IB, JB, IC, JC, ID, JD
   real(4) :: alpha, beta  ! Any compatible kind
   real(4), device, dimension(*) :: A, B, C, D  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC, descrD
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.41. cublasMpMatmul

This is the multi-processor version of the matrix multiplication operation.

Matmul performs one of the matrix-matrix operations

D := alpha*op(A)*op(B) + beta*C,

where op(X) is one of

op(X) = X or op(X) = X**T, as set by a call to `cublasMpMatmulDescriptorSetAttribute()`.

alpha and beta are scalars, and A, B, C, and D are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C and D are m by n matrices. The data for A, B, C, and D should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB, descrC, and descrD via the `cublasMpMatrixDescriptorCreate()` function. The datatype is also specified there. M, N, and K are the global matrix dimensions. IA, JA, IB, JB, IC, JC, ID, and JD are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpMatmul(handle, matmulDescr, M, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   D, ID, JD, descrD, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   type(cublasMpMatmulDescriptor) :: matmulDescr
   integer(8), intent(in) :: M, N, K, IA, JA, IB, JB, IC, JC, ID, JD
   real(4) :: alpha, beta  ! Any supported type and kind
   real(4), device, dimension(*) :: A, B, C, D  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC, descrD
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceInBytes)  ! Any type
```

9.9.42. cublasMpSyrk_bufferSize

This functions computes the workspace requirements of cublasMpSyrk.

```
type(cublasMpStatus) function cublasMpSyrk_bufferSize(handle, uplo, trans, &
   N, K, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.43. cublasMpSyrk

This is the multi-processor version of the BLAS SYRK operation, similar to the ScaLAPACK PBLAS functions pdsyrk, pzsyrk, etc.

SYRK performs one of the symmetric rank k operations

C := alpha*A*A**T + beta*C, or

C := alpha*A**T*A + beta*C

alpha and beta are scalars, and A and C are matrices. A is either N x K or K x N depending on the trans argument, and C is N x N. The data for A and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpSyrk(handle, uplo, trans, &
   N, K, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.44. cublasMpSyr2k_bufferSize

This functions computes the workspace requirements of cublasMpSyr2k.

```
type(cublasMpStatus) function cublasMpSyr2k_bufferSize(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.45. cublasMpSyr2k

This is the multi-processor version of the BLAS SYR2K operation, similar to the ScaLAPACK PBLAS functions pdsyr2k, pzsyr2k, etc.

SYR2K performs one of the symmetric rank 2k operations

C := alpha*A*B**T + alpha*B*A**T + beta*C, or

C := alpha*A**T*B + alpha*B**T*A + beta*C

where alpha and beta are scalars, C is an N x N symmetric matrix, and A and B are either N x K or K x N depending on the trans argument.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpSyr2k(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.46. cublasMpSyrkx_bufferSize

This functions computes the workspace requirements of cublasMpSyrkx.

```
type(cublasMpStatus) function cublasMpSyrkx_bufferSize(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.47. cublasMpSyrkx

This is the multi-processor version of the BLAS SYRKX extension, similar to the ScaLAPACK PBLAS functions pdsyrkx, pzsyrkx, etc.

SYRKX performs a symmetric rank k update

C := alpha*A*B**T + beta*C, or

C := alpha*A**T*B + beta*C

where alpha and beta are scalars, C is an N x N symmetric matrix, and A and B are either N x K or K x N depending on the trans argument. Unlike SYR2K only one product term is formed.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpSyrkx(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.48. cublasMpSymm_bufferSize

This functions computes the workspace requirements of cublasMpSymm.

```
type(cublasMpStatus) function cublasMpSymm_bufferSize(handle, side, uplo, M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.49. cublasMpSymm

This is the multi-processor version of the BLAS SYMM operation, similar to the ScaLAPACK PBLAS functions pdsymm, pzsymm, etc.

SYMM performs one of the matrix-matrix operations

C := alpha*A*B + beta*C, or

C := alpha*B*A + beta*C

where alpha and beta are scalars, A is a symmetric matrix and B and C are M x N matrices. The side argument selects whether A multiplies from the left or the right, and uplo selects which triangle of A is referenced.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpSymm(handle, side, uplo, M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   real(4) :: alpha, beta  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.50. cublasMpTrsm_bufferSize

This functions computes the workspace requirements of cublasMpTrsm.

```
type(cublasMpStatus) function cublasMpTrsm_bufferSize(handle, side, uplo, trans, diag,
↪ &
   M, N, alpha, A, IA, JA, descrA, B, IB, JB, descrB, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo, trans, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4) :: alpha  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.51. cublasMpTrsm

This is the multi-processor version of the BLAS TRSM operation, similar to the ScaLAPACK PBLAS functions pdtrsm, pztrsm, etc.

TRSM solves one of the matrix equations

op(A)*X = alpha*B, or

X*op(A) = alpha*B

alpha is a scalar, A and B are matrices whose dimensions are determined by the side argument. The data for A and B should be properly distributed over the process grid. That mapping is contained within the descriptors descrA and descrB via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IB, and JB are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpTrsm(handle, side, uplo, trans, diag,  &
   M, N, alpha, A, IA, JA, descrA, B, IB, JB, descrB, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo, trans, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB
   real(4) :: alpha  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)     ! Any type
```

9.9.52. cublasMpTrmm_bufferSize

This functions computes the workspace requirements of cublasMpTrmm.

```
type(cublasMpStatus) function cublasMpTrmm_bufferSize(handle, side, uplo, trans, diag,
↪ M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo, trans, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   real(4) :: alpha  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.53. cublasMpTrmm

This is the multi-processor version of the BLAS TRMM operation, similar to the ScaLAPACK PBLAS functions pdtrmm, pztrmm, etc.

TRMM performs one of the matrix-matrix operations

C := alpha*op(A)*B, or

C := alpha*B*op(A)

where alpha is a scalar, A is a triangular matrix, and B and C are M x N matrices. op(A) is selected by the trans argument, side selects whether op(A) multiplies from the left or the right, uplo selects which triangle of A is referenced, and diag states whether A is unit triangular. There is no beta argument; the result is written to a separate output matrix C.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpTrmm(handle, side, uplo, trans, diag, M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo, trans, diag
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   real(4) :: alpha  ! type and kind compatible with computeType
   real(4), device, dimension(*) :: A, B, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.54. cublasMpGeadd_bufferSize

This functions computes the workspace requirements of cublasMpGeadd.

```
type(cublasMpStatus) function cublasMpGeadd_bufferSize(handle, trans, &
   M, N, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: trans
   integer(8), intent(in) :: M, N, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Any compatible kind
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.55. cublasMpGeadd

This is the multi-processor version of a general matrix addition function.

GEADD performs the matrix-matrix addition operation

C := alpha*A + beta*C

alpha and beta are scalars, and A and C are matrices. A is either M x N or N x M depending on the trans argument, and C is M x N. The data for A and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpGeadd(handle, trans, &
   M, N, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: trans
   integer(8), intent(in) :: M, N, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Any compatible type and kind
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.56. cublasMpTradd_bufferSize

This functions computes the workspace requirements of cublasMpTradd.

```
type(cublasMpStatus) function cublasMpTradd_bufferSize(handle, uplo, trans, &
   M, N, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: M, N, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Any compatible kind
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.57. cublasMpTradd

This is the multi-processor version of a trapezoidal matrix addition function.

TRADD performs the trapezoidal matrix-matrix addition operation

C := alpha*A + beta*C

alpha and beta are scalars, and A and C are matrices. A is either M x N or N x M depending on the trans argument, and C is M x N. The data for A and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpTradd(handle, uplo, trans, &
   M, N, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: M, N, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Any compatible type and kind
   real(4), device, dimension(*) :: A, C  ! Any supported type and kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.58. cublasMpHemm_bufferSize

This functions computes the workspace requirements of cublasMpHemm.

```
type(cublasMpStatus) function cublasMpHemm_bufferSize(handle, side, uplo, M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha, beta  ! Any compatible complex kind
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.59. cublasMpHemm

This is the multi-processor version of the BLAS HEMM operation, similar to the ScaLAPACK PBLAS functions pchemm, pzhemm, etc.

HEMM performs one of the matrix-matrix operations

C := alpha*A*B + beta*C, or

C := alpha*B*A + beta*C

where alpha and beta are complex scalars, A is a Hermitian matrix and B and C are M x N complex matrices. The side argument selects whether A multiplies from the left or the right, and uplo selects which triangle of A is referenced. This operation is defined for complex data only.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. M and N are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpHemm(handle, side, uplo, M, N, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: side, uplo
   integer(8), intent(in) :: M, N, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha, beta  ! Any compatible complex kind
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.60. cublasMpHer2k_bufferSize

This functions computes the workspace requirements of cublasMpHer2k.

```
type(cublasMpStatus) function cublasMpHer2k_bufferSize(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha  ! Complex; kind varies with compute type
   real(4)    :: beta   ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.61. cublasMpHer2k

This is the multi-processor version of the BLAS HER2K operation, similar to the ScaLAPACK PBLAS functions pcher2k, pzher2k, etc.

HER2K performs one of the Hermitian rank 2k operations

C := alpha*A*B**H + conjg(alpha)*B*A**H + beta*C, or

C := alpha*A**H*B + conjg(alpha)*B**H*A + beta*C

where alpha is a complex scalar, beta is a real scalar, and C is an N x N Hermitian matrix. Because C is Hermitian the imaginary part of its diagonal is zero on return. This operation is defined for complex data only.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpHer2k(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha  ! Complex; kind varies with compute type
   real(4)    :: beta   ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.62. cublasMpHerkx_bufferSize

This functions computes the workspace requirements of cublasMpHerkx.

```
type(cublasMpStatus) function cublasMpHerkx_bufferSize(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha  ! Complex; kind varies with compute type
   real(4)    :: beta   ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.63. cublasMpHerkx

This is the multi-processor version of the BLAS HERKX extension, similar to the ScaLAPACK PBLAS functions pcherkx, pzherkx, etc.

HERKX performs a Hermitian rank k update using two input matrices

C := alpha*A*B**H + beta*C, or

C := alpha*A**H*B + beta*C

where alpha is a complex scalar, beta is a real scalar, and C is an N x N Hermitian matrix. Unlike HER2K only one product term is formed. Because C is Hermitian the imaginary part of its diagonal is zero on return, even though alpha*A*B**H is not Hermitian in general. This operation is defined for complex data only.

The data for A, B and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA, descrB and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IB, JB, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpHerkx(handle, uplo, trans, N, K, &
   alpha, A, IA, JA, descrA, B, IB, JB, descrB, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IB, JB, IC, JC
   complex(4) :: alpha  ! Complex; kind varies with compute type
   real(4)    :: beta   ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, B, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrB, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.64. cublasMpHerk_bufferSize

This functions computes the workspace requirements of cublasMpHerk.

```
type(cublasMpStatus) function cublasMpHerk_bufferSize(handle, uplo, trans, &
   N, K, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   computeType, devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(out) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
```

9.9.65. cublasMpHerk

This is the multi-processor version of the BLAS HERK operation, similar to the ScaLAPACK PBLAS functions pcherk, pzherk, etc.

HERK performs one of the Hermitian rank k operations

C := alpha*A*A**H + beta*C, or

C := alpha*A**H*A + beta*C

where alpha and beta are real scalars and C is an N x N Hermitian matrix. A is either N x K or K x N depending on the trans argument. Because C is Hermitian the imaginary part of its diagonal is zero on return. This operation is defined for complex data only, but note that both alpha and beta are real.

The data for A and C should be properly distributed over the process grid. That mapping is contained within the descriptors descrA and descrC via the cublasMpMatrixDescriptorCreate() function. The datatype is also specified then. N and K are the global matrix dimensions. IA, JA, IC, and JC are 1-based, and typically equal to 1 for a full matrix. Integer(4) input values will be promoted to integer(8) according to the interface.

```
type(cublasMpStatus) function cublasMpHerk(handle, uplo, trans, &
   N, K, alpha, A, IA, JA, descrA, beta, C, IC, JC, descrC, &
   computeType, bufferOnDevice, devWorkspaceSizeInBytes, &
   bufferOnHost, hostWorkspaceSizeInBytes)
   type(cublasMpHandle) :: handle
   integer(4) :: uplo, trans
   integer(8), intent(in) :: N, K, IA, JA, IC, JC
   real(4) :: alpha, beta  ! Real-valued; kind varies with compute type
   complex(4), device, dimension(*) :: A, C  ! Any supported complex kind
   type(cublasMpMatrixDescriptor) :: descrA, descrC
   type(cublasComputeType) :: computeType
   integer(8), intent(in) :: devWorkspaceSizeInBytes, hostWorkspaceSizeInBytes
   integer(1), device :: bufferOnDevice(devWorkspaceSizeInBytes) ! Any type
   integer(1)         :: bufferOnHost(hostWorkspaceSizeInBytes)  ! Any type
```

9.9.66. cublasMpLoggerSetFile

This function specifies the Fortran unit to be used as the cublasMp logfile.

```
type(cublasMpStatus) function cublasMpLoggerSetFile(unit)
  integer :: unit
```

9.9.67. cublasMpLoggerOpenFile

This function specifies a Fortran character string to be opened and used as the cublasMp logfile.

```
type(cublasMpStatus) function cublasMpLoggerOpenFile(logFile)
  character*(*) :: logFile
```

9.9.68. cublasMpLoggerSetLevel

This function specifies the cublasMp logging level.

```
type(cublasMpStatus) function cublasMpLoggerSetLevel(level)
  integer :: level
```

9.9.69. cublasMpLoggerSetMask

This function specifies the cublasMp logging mask.

```
type(cublasMpStatus) function cublasMpLoggerSetMask(mask)
  integer :: mask
```

9.9.70. cublasMpLoggerForceDisable

This function disables cublasMp logging.

```
type(cublasMpStatus) function cublasMpLoggerForceDisable()
```

9.9.71. cublasMpLoggerSetCallback

This function installs a logging callback. Pass `c_null_funptr` to clear a previously registered callback.

```
type(cublasMpStatus) function cublasMpLoggerSetCallback(callback)
   type(c_funptr) :: callback
```

Chapter 10. FFT Runtime Library APIs

This section describes the Fortran interfaces to the cuFFT library. The FFT functions are only accessible from host code. All of the runtime API routines are integer functions that return an error code; they return a value of CUFFT_SUCCESS if the call was successful, or another cuFFT status return value if there was an error.

Chapter 10 contains examples of accessing the cuFFT library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line

```
use cufft
```

to your program unit.

Beginning with our 21.9 release, we also support a cufftXt module, which provides interfaces to the multi-gpu support available in the cuFFT library. These interfaces can be used within any Fortran program by adding the line

```
use cufftxt
```

to your program unit. The cufftXt interfaces are documented beginning in section 4 of this chapter.

Unless a specific kind is provided in the following interfaces, the plain integer type implies integer(4) and the plain real type implies real(4).

10.1. CUFFT Definitions and Helper Functions

This section contains definitions and data types used in the cuFFT library and interfaces to the cuFFT helper functions.

The cuFFT module contains the following constants and enumerations:

```
integer, parameter :: CUFFT_FORWARD = -1
integer, parameter :: CUFFT_INVERSE = 1
```

```
! CUFFT Status
enum, bind(C)
    enumerator :: CUFFT_SUCCESS           = 0
    enumerator :: CUFFT_INVALID_PLAN      = 1
    enumerator :: CUFFT_ALLOC_FAILED      = 2
    enumerator :: CUFFT_INVALID_TYPE      = 3
    enumerator :: CUFFT_INVALID_VALUE     = 4
    enumerator :: CUFFT_INTERNAL_ERROR    = 5
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CUFFT_EXEC_FAILED        = 6
    enumerator :: CUFFT_SETUP_FAILED       = 7
    enumerator :: CUFFT_INVALID_SIZE       = 8
    enumerator :: CUFFT_UNALIGNED_DATA     = 9
    enumerator :: CUFFT_INCOMPLETE_PARAMETER_LIST = 10 ! CUDA 12.x only
    enumerator :: CUFFT_INVALID_DEVICE     = 11
    enumerator :: CUFFT_PARSE_ERROR        = 12        ! CUDA 12.x only
    enumerator :: CUFFT_NO_WORKSPACE       = 13
    enumerator :: CUFFT_NOT_IMPLEMENTED    = 14
    enumerator :: CUFFT_LICENSE_ERROR      = 15        ! CUDA 12.x only
    enumerator :: CUFFT_NOT_SUPPORTED      = 16
    enumerator :: CUFFT_MISSING_DEPENDENCY = 17        ! CUDA 13.x only
    enumerator :: CUFFT_NVRTC_FAILURE      = 18        ! CUDA 13.x only
    enumerator :: CUFFT_NVJITLINK_FAILURE  = 19        ! CUDA 13.x only
    enumerator :: CUFFT_NVSHMEM_FAILURE    = 20        ! CUDA 13.x only
end enum
```

```
! CUFFT Transform Types
enum, bind(C)
    enumerator :: CUFFT_R2C = z'2a'     ! Real to Complex (interleaved)
    enumerator :: CUFFT_C2R = z'2c'     ! Complex (interleaved) to Real
    enumerator :: CUFFT_C2C = z'29'     ! Complex to Complex, interleaved
    enumerator :: CUFFT_D2Z = z'6a'     ! Double to Double-Complex
    enumerator :: CUFFT_Z2D = z'6c'     ! Double-Complex to Double
    enumerator :: CUFFT_Z2Z = z'69'     ! Double-Complex to Double-Complex
end enum
```

```
! CUFFT Data Layouts
enum, bind(C)
    enumerator :: CUFFT_COMPATIBILITY_FFTW_PADDING    = 1
end enum
```

```
integer, parameter :: CUFFT_COMPATIBILITY_DEFAULT = CUFFT_COMPATIBILITY_FFTW_PADDING
```

10.1.1. cufftSetStream

This function sets the stream to be used by the cuFFT library to execute its routines.

```
integer(4) function cufftSetStream(plan, stream)
  integer :: plan
  integer(kind=cuda_stream_kind) :: stream
```

10.1.2. cufftGetVersion

This function returns the version number of cuFFT.

```
integer(4) function cufftGetVersion( version )
  integer :: version
```

10.1.3. cufftGetProperty

This function returns in `pvalue` the value for the property described by `ptype` of the dynamically linked cuFFT library.

```
integer(4) function cufftGetProperty( ptype, pvalue )
  integer(4) :: ptype
  integer(4) :: pvalue
```

10.1.4. cufftSetPlanPropertyInt64

This function sets a plan property of type `integer(8)`. The `property` argument is an enumerator of the following type:

```
enum, bind(C)
    enumerator :: NVFFT_PLAN_PROPERTY_INT64_PATIENT_JIT = 1
    enumerator :: NVFFT_PLAN_PROPERTY_INT64_MAX_NUM_HOST_THREADS = 2
end enum
```

```
integer(4) function cufftSetPlanPropertyInt64( plan, property, inputValueInt )
  integer(4) :: plan
  integer(4) :: property
  integer(8) :: inputValueInt
```

10.1.5. cufftGetPlanPropertyInt64

This function gets a plan property of type `integer(8)`.

```
integer(4) function cufftGetPlanPropertyInt64( plan, property, returnPtrValue )
  integer(4) :: plan
  integer(4) :: property
  integer(8) :: returnPtrValue
```

10.1.6. cufftResetPlanProperty

This function resets a plan property to its default value.

```
integer(4) function cufftResetPlanProperty( plan, property )
  integer(4) :: plan
  integer(4) :: property
```

10.1.7. cufftSetAutoAllocation

This function indicates that the caller intends to allocate and manage work areas for plans that have been generated. cuFFT default behavior is to allocate the work area at plan generation time. If cufftSetAutoAllocation() has been called with autoAllocate set to 0 prior to one of the cufftMakePlan*() calls, cuFFT does not allocate the work area. This is the preferred sequence for callers wishing to manage work area allocation.

```
integer(4) function cufftSetAutoAllocation(plan, autoAllocate)
  integer(4) :: plan, autoallocate
```

10.1.8. cufftSetWorkArea

This function overrides the work area pointer associated with a plan. If the work area was auto-allocated, cuFFT frees the auto-allocated space. The cufftExecute*() calls assume that the work area pointer is valid and that it points to a contiguous region in device memory that does not overlap with any other work area. If this is not the case, results are indeterminate.

```
integer(4) function cufftSetWorkArea(plan, workArea)
  integer(4) :: plan
  integer, device :: workArea(*) ! Can be integer, real, complex
                                 ! or a type(c_devptr)
```

10.1.9. cufftDestroy

This function frees all GPU resources associated with a cuFFT plan and destroys the internal plan data structure.

```
integer(4) function cufftDestroy( plan )
  integer :: plan
```

10.2. CUFFT Plans and Estimated Size Functions

This section contains functions from the cuFFT library used to create plans and estimate work buffer size.

10.2.1. cufftPlan1d

This function creates a 1D FFT plan configuration for a specified signal size and data type. Nx is the size of the transform; batch is the number of transforms of size nx.

```
integer(4) function cufftPlan1d(plan, nx, ffttype, batch)
  integer :: plan
  integer :: nx
  integer :: ffttype
  integer :: batch
```

10.2.2. cufftPlan2d

This function creates a 2D FFT plan configuration according to a specified signal size and data type. For a Fortran array(nx,ny), nx is the size of the of the 1st dimension in the transform, but the 2nd size argument to the function; ny is the size of the 2nd dimension, and the 1st size argument to the function.

```
integer(4) function cufftPlan2d( plan, ny, nx, ffttype )
  integer :: plan
  integer :: ny, nx
  integer :: ffttype
```

10.2.3. cufftPlan3d

This function creates a 3D FFT plan configuration according to a specified signal size and data type. For a Fortran array(nx,ny,nz), nx is the size of the of the 1st dimension in the transform, but the 3rd size argument to the function; nz is the size of the 3rd dimension, and the 1st size argument to the function.

```
integer(4) function cufftPlan3d( plan, nz, ny, nx, ffttype )
  integer :: plan
  integer :: nz, ny, nx
  integer :: ffttype
```

10.2.4. cufftPlanMany

This function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. Batch is the number of transforms to configure. This function supports more complicated input and output data layouts using the arguments inembed, istride, idist, onembed, ostride, and odist. In the C function, if inembed and onembed are set to NULL, all other stride information is ignored. Fortran programmers can pass NULL when using the NVIDIA cufft module by setting an F90 pointer to null(), either through direct assignment, using c_f_pointer() with c_null_ptr as the first argument, or the nullify statement, then passing the nullified F90 pointer as the actual argument for the inembed and onembed dummies.

```
integer(4) function cufftPlanMany(plan, rank, n, inembed, istride, idist, onembed,
↪ostride, odist, ffttype, batch )
  integer :: plan
  integer :: rank
  integer :: n
  integer :: inembed, onembed
  integer :: istride, idist, ostride, odist
  integer :: ffttype, batch
```

10.2.5. cufftCreate

This function creates an opaque handle for further cuFFT calls and allocates some small data structures on the host. In C, the handle type is currently typedef'ed to an int, so in Fortran we use an integer*4 to hold the plan.

```
integer(4) function cufftCreate(plan)
  integer(4) :: plan
```

10.2.6. cufftMakePlan1d

Following a call to cufftCreate(), this function creates a 1D FFT plan configuration for a specified signal size and data type. Nx is the size of the transform; batch is the number of transforms of size nx. If `cufftXtSetGPUs` was called prior to this call with multiple GPUs, then `workSize` is an array containing multiple sizes. The workSize values are in bytes.

```
integer(4) function cufftMakePlan1d(plan, nx, ffttype, batch, worksize)
  integer(4) :: plan
  integer(4) :: nx
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.7. cufftMakePlan2d

Following a call to cufftCreate(), this function creates a 2D FFT plan configuration according to a specified signal size and data type. For a Fortran array(nx,ny), nx is the size of the of the 1st dimension in the transform, but the 2nd size argument to the function; ny is the size of the 2nd dimension, and the 1st size argument to the function. If `cufftXtSetGPUs` was called prior to this call with multiple GPUs, then `workSize` is an array containing multiple sizes. The workSize values are in bytes.

```
integer(4) function cufftMakePlan2d(plan, ny, nx, ffttype, workSize)
  integer(4) :: plan
  integer(4) :: ny, nx
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.8. cufftMakePlan3d

Following a call to cufftCreate(), this function creates a 3D FFT plan configuration according to a specified signal size and data type. For a Fortran array(nx,ny,nz), nx is the size of the of the 1st dimension in the transform, but the 3rd size argument to the function; nz is the size of the 3rd dimension, and the 1st size argument to the function. If `cufftXtSetGPUs` was called prior to this call with multiple GPUs, then `workSize` is an array containing multiple sizes. The workSize values are in bytes.

```
integer(4) function cufftMakePlan3d(plan, nz, ny, nx, ffttype, workSize)
  integer(4) :: plan
  integer(4) :: nz, ny, nx
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.9. cufftMakePlanMany

Following a call to cufftCreate(), this function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. Batch is the number of transforms to configure. This function supports more complicated input and output data layouts using the arguments inembed, istride, idist, onembed, ostride, and odist.

In the C function, if inembed and onembed are set to NULL, all other stride information is ignored. Fortran programmers can pass NULL when using the NVIDIA cufft module by setting an F90 pointer to null(), either through direct assignment, using c_f_pointer() with c_null_ptr as the first argument, or the nullify statement, then passing the nullified F90 pointer as the actual argument for the inembed and onembed dummies.

If `cufftXtSetGPUs` was called prior to this call with multiple GPUs, then `workSize` is an array containing multiple sizes. The workSize values are in bytes.

```
integer(4) function cufftMakePlanMany(plan, rank, n, inembed, istride, idist, onembed,
↪ ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan
  integer(4) :: rank
  integer :: n(rank)
  integer :: inembed(rank), onembed(rank)
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: istride, idist, ostride, odist
  integer(4) :: ffttype, batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.10. cufftMakePlanMany64

Following a call to cufftCreate(), this function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. This is the 64-bit version of `cufftMakePlanMany`, which uses `integer(8)` types for the array dimensions, strides, distances, and batch count parameters.

```
integer(4) function cufftMakePlanMany64(plan, rank, n, inembed, istride, idist,
↪onembed, ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan
  integer(4) :: rank
  integer(8) :: n(rank)
  integer(8) :: inembed(rank), onembed(rank)
  integer(8) :: istride, idist, ostride, odist
  integer(4) :: ffttype
  integer(8) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.11. cufftEstimate1d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate1d(nx, ffttype, batch, workSize)
  integer(4) :: nx
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.12. cufftEstimate2d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate2d(ny, nx, ffttype, workSize)
  integer(4) :: ny, nx
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.13. cufftEstimate3d

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimate3d(nz, ny, nx, ffttype, workSize)
  integer(4) :: nz, ny, nx
  integer(4) :: ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.14. cufftEstimateMany

This function returns an estimate for the size of the work area required, in bytes, given the specified size and data type, and assuming default plan settings.

```
integer(4) function cufftEstimateMany(rank, n, inembed, istride, idist, onembed,
↪ostride, odist, ffttype, batch, workSize)
  integer(4) :: rank, istride, idist, ostride, odist
  integer(4), dimension(rank) :: n, inembed, onembed
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.15. cufftGetSize1d

This function gives a more accurate estimate than cufftEstimate1d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize1d(plan, nx, ffttype, batch, workSize)
  integer(4) :: plan, nx, ffttype, batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.16. cufftGetSize2d

This function gives a more accurate estimate than cufftEstimate2d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize2d(plan, ny, nx, ffttype, workSize)
  integer(4) :: plan, ny, nx, ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.17. cufftGetSize3d

This function gives a more accurate estimate than cufftEstimate3d() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSize3d(plan, nz, ny, nx, ffttype, workSize)
  integer(4) :: plan, nz, ny, nx, ffttype
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.18. cufftGetSizeMany

This function gives a more accurate estimate than cufftEstimateMany() of the size of the work area required, in bytes, given the specified plan parameters and taking into account any plan settings which may have been made.

```
integer(4) function cufftGetSizeMany(plan, rank, n, inembed, istride, idist, onembed,
↪ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan, rank, istride, idist, ostride, odist
  integer(4), dimension(rank) :: n, inembed, onembed
  integer(4) :: ffttype
  integer(4) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.19. cufftGetSizeMany64

This is the 64-bit version of `cufftGetSizeMany`. It gives a more accurate estimate of the size of the work area required, in bytes, using `integer(8)` types for the array dimensions, strides, distances, and batch count parameters.

```
integer(4) function cufftGetSizeMany64(plan, rank, n, inembed, istride, idist,
↪onembed, ostride, odist, ffttype, batch, workSize)
  integer(4) :: plan
  integer(4) :: rank
  integer(8) :: n(rank)
  integer(8) :: inembed(rank), onembed(rank)
  integer(8) :: istride, idist, ostride, odist
  integer(4) :: ffttype
  integer(8) :: batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.2.20. cufftGetSize

Once plan generation has been done, either with the original API or the extensible API, this call returns the actual size of the work area required, in bytes, to support the plan. Callers who choose to manage work area allocation within their application must use this call after plan generation, and after any cufftSet*() calls subsequent to plan generation, if those calls might alter the required work space size.

```
integer(4) function cufftGetSize(plan, workSize)
  integer(4) :: plan
  integer(kind=int_ptr_kind()) :: workSize(*)
```

10.3. CUFFT Execution Functions

This section contains the execution functions, which perform the actual Fourier transform, in the cuFFT library.

10.3.1. cufftExecC2C

This function executes a single precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecC2C( plan, idata, odata, direction )
  integer :: plan
  complex(4), device, dimension(*) :: idata, odata
  integer :: direction
```

10.3.2. cufftExecR2C

This function executes a single precision real-to-complex, implicity forward, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform, but note there are data layout differences between in-place and out-of-place transforms for real-to- complex FFTs in cuFFT.

```
integer(4) function cufftExecR2C( plan, idata, odata )
  integer :: plan
  real(4), device, dimension(*) :: idata
  complex(4), device, dimension(*) :: odata
```

10.3.3. cufftExecC2R

This function executes a single precision complex-to-real, implicity inverse, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecC2R( plan, idata, odata )
  integer :: plan
  complex(4), device, dimension(*) :: idata
  real(4), device, dimension(*) :: odata
```

10.3.4. cufftExecZ2Z

This function executes a double precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecZ2Z( plan, idata, odata, direction )
  integer :: plan
  complex(8), device, dimension(*) :: idata, odata
  integer :: direction
```

10.3.5. cufftExecD2Z

This function executes a double precision real-to-complex, implicity forward, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform, but note there are data layout differences between in-place and out-of-place transforms for real-to- complex FFTs in cuFFT.

```
integer(4) function cufftExecD2Z( plan, idata, odata )
  integer :: plan
  real(8), device, dimension(*) :: idata
  complex(8), device, dimension(*) :: odata
```

10.3.6. cufftExecZ2D

This function executes a double precision complex-to-real, implicity inverse, cuFFT transform plan. If idata and odata are the same, this function does an in-place transform.

```
integer(4) function cufftExecZ2D( plan, idata, odata )
  integer :: plan
  complex(8), device, dimension(*) :: idata
  real(8), device, dimension(*) :: odata
```

10.4. CUFFTXT Definitions and Helper Functions

This section contains definitions and data types used in the cufftXt library and interfaces to helper functions. Beginning with NVHPC version 22.5, this module also contains some interfaces and definitions used with the cuFFTMp library.

The cufftXt module contains the following constants and enumerations:

```
integer, parameter :: MAX_CUDA_DESCRIPTOR_GPUS = 64
```

```
! libFormat enum is used for the library member of cudaLibXtDesc
enum, bind(C)
    enumerator :: LIB_FORMAT_CUFFT     = 0
    enumerator :: LIB_FORMAT_UNDEFINED = 1
end enum
```

```
! cufftXtSubFormat identifies the data layout of a memory descriptor
enum, bind(C)
    ! by default input is in linear order across GPUs
    enumerator :: CUFFT_XT_FORMAT_INPUT = 0

    ! by default output is in scrambled order depending on transform
    enumerator :: CUFFT_XT_FORMAT_OUTPUT = 1

    ! by default inplace is input order, which is linear across GPUs
    enumerator :: CUFFT_XT_FORMAT_INPLACE = 2

    ! shuffled output order after execution of the transform
    enumerator :: CUFFT_XT_FORMAT_INPLACE_SHUFFLED = 3

    ! shuffled input order prior to execution of 1D transforms
    enumerator :: CUFFT_XT_FORMAT_1D_INPUT_SHUFFLED = 4

    ! distributed input order
    enumerator :: CUFFT_XT_FORMAT_DISTRIBUTED_INPUT = 5

    ! distributed output order
    enumerator :: CUFFT_XT_FORMAT_DISTRIBUTED_OUTPUT = 6

    enumerator :: CUFFT_FORMAT_UNDEFINED = 7
end enum
```

```
! cufftXtCopyType specifies the type of copy for cufftXtMemcpy
enum, bind(C)
    enumerator :: CUFFT_COPY_HOST_TO_DEVICE   = 0
    enumerator :: CUFFT_COPY_DEVICE_TO_HOST   = 1
    enumerator :: CUFFT_COPY_DEVICE_TO_DEVICE = 2
    enumerator :: CUFFT_COPY_UNDEFINED        = 3
end enum
```

```
! cufftXtQueryType specifies the type of query for cufftXtQueryPlan
enum, bind(c)
    enumerator :: CUFFT_QUERY_1D_FACTORS = 0
    enumerator :: CUFFT_QUERY_UNDEFINED  = 1
end enum
```

```
! cufftXtWorkAreaPolicy specifies the policy for cufftXtSetWorkAreaPolicy
enum, bind(c)
    enumerator :: CUFFT_WORKAREA_MINIMAL     = 0 ! maximum reduction
    enumerator :: CUFFT_WORKAREA_USER        = 1 ! use workSize parameter as limit
    enumerator :: CUFFT_WORKAREA_PERFORMANCE = 2 ! default - 1x overhead or more, max
↪perf
end enum
```

```
! cufftMpCommType specifies how to initialize cuFFTMp
enum, bind(c)
    enumerator :: CUFFT_COMM_MPI  = 0
    enumerator :: CUFFT_COMM_NONE = 1
end enum
```

```
! cufftXtCallbackType specifies the type of callback
enum, bind(c)
    enumerator :: CUFFT_CB_LD_COMPLEX         = 0
    enumerator :: CUFFT_CB_LD_COMPLEX_DOUBLE  = 1
    enumerator :: CUFFT_CB_LD_REAL            = 2
    enumerator :: CUFFT_CB_LD_REAL_DOUBLE     = 3
    enumerator :: CUFFT_CB_ST_COMPLEX         = 4
    enumerator :: CUFFT_CB_ST_COMPLEX_DOUBLE  = 5
    enumerator :: CUFFT_CB_ST_REAL            = 6
    enumerator :: CUFFT_CB_ST_REAL_DOUBLE     = 7
    enumerator :: CUFFT_CB_UNDEFINED          = 8
end enum
```

The cufftXt module contains the following derived type definitions:

```
! cufftXt1dFactors type
type, bind(c) :: cufftXt1dFactors
    integer(8) :: size
    integer(8) :: stringCount
    integer(8) :: stringLength
    integer(8) :: subStringLength
    integer(8) :: factor1
    integer(8) :: factor2
    integer(8) :: stringMask
    integer(8) :: subStringMask
    integer(8) :: factor1Mask
    integer(8) :: factor2Mask
    integer(4) :: stringShift
    integer(4) :: subStringShift
    integer(4) :: factor1Shift
    integer(4) :: factor2Shift
end type cufftXt1dFactors
```

```
type, bind(C) :: cudaXtDesc
    integer(4) :: version
    integer(4) :: nGPUs
    integer(4) :: GPUs(MAX_CUDA_DESCRIPTOR_GPUS)
    type(c_devptr) :: data(MAX_CUDA_DESCRIPTOR_GPUS)
    integer(8) :: size(MAX_CUDA_DESCRIPTOR_GPUS)
    type(c_ptr) :: cudaXtState
end type cudaXtDesc
```

```
type, bind(C) :: cudaLibXtDesc
    integer(4) :: version
    type(c_ptr) :: descriptor     ! cudaXtDesc *descriptor
    integer(4) :: library         ! libFormat library
    integer(4) :: subFormat
    type(c_ptr) :: libDescriptor  ! void *libDescriptor
end type cudaLibXtDesc
```

```
type, bind(C) :: cufftBox3d
    integer(8) :: lower(3)
    integer(8) :: upper(3)
    integer(8) :: strides(3)
end type cufftBox3d
```

10.4.1. cufftXtSetGPUs

This function identifies which GPUs are to be used with the plan. The call to `cufftXtSetGPUs` must occur after the call to `cufftCreate` but before the call to `cufftMakePlan*`.

```
integer(4) function cufftXtSetGPUs( plan, nGPUs, whichGPUs )
  integer(4) :: plan
  integer(4) :: nGPUs
  integer(4) :: whichGPUs(*)
```

10.4.2. cufftXtMalloc

This function allocates a cufftXt descriptor, and memory for data in the GPUs associated with the plan. The value of `cufftXtSubFormat` determines if the buffer will be used for input or output. Fortran programmers should declare and pass a pointer to a `type(cudaLibXtDesc)` variable so the entire information can be stored, and also freed in subsequent calls to `cufftXtFree`. For programmers comfortable with the C interface, a variant of this function can take a `type(c_ptr)` for the 2nd argument.

```
integer(4) function cufftXtMalloc( plan, descriptor, format )
  integer(4) :: plan
  type(cudaLibXtDesc), pointer :: descriptor  ! A type(c_ptr) is also accepted.
  integer(4) :: format ! cufftXtSubFormat value
```

10.4.3. cufftXtFree

This function frees the cufftXt descriptor, and all memory associated with it. The descriptor and memory must have been allocated by a previous call to `cufftXtMalloc`. Fortran programmers should declare and pass a pointer to a `type(cudaLibXtDesc)` variable. For programmers comfortable with the C interface, a variant of this function can take a `type(c_ptr)` as the only argument.

```
integer(4) function cufftXtFree( descriptor )
  type(cudaLibXtDesc), pointer :: descriptor  ! A type(c_ptr) is also accepted.
```

10.4.4. cufftXtMemcpy

This function copies data between buffers on the host and GPUs, or between GPUs. The value of the `type` argument determines the copy direction. In addition, this Fortran function is overloaded to take a `type(cudaLibXtDesc)` variable for the destination (H2D transfer), for the source (D2H transfer), or for both (D2D transfer), in which case the `type` argument is not required.

```
integer(4) function cufftXtMemcpy( plan, dst, src, type )
  integer(4) :: plan
  type(cudaLibXtDesc) :: dst  ! Or any host buffer, depending on the type
  type(cudaLibXtDesc) :: src  ! Or any host buffer, depending on the type
  integer(4) :: type          ! optional cufftXtCopyType value
```

10.5. CUFFTXT Plans and Work Area Functions

This section contains functions from the cufftXt library used to create plans and manage work buffers.

10.5.1. cufftXtMakePlanMany

Following a call to cufftCreate(), this function creates an FFT plan configuration of dimension rank, with sizes specified in the array n. Batch is the number of transforms to configure. This function supports more complicated input and output data layouts using the arguments inembed, istride, idist, onembed, ostride, and odist. In the C function, if inembed and onembed are set to NULL, all other stride information is ignored. Fortran programmers can pass NULL when using the NVIDIA cufft module by setting an F90 pointer to null(), either through direct assignment, using c_f_pointer() with c_null_ptr as the first argument, or the nullify statement, then passing the nullified F90 pointer as the actual argument for the inembed and onembed dummies.

```
integer(4) function cufftXtMakePlanMany(plan, rank, n, inembed, istride, &
    idist, inputType, onembed, ostride, odist, outputType, batch, workSize, &
    executionType)
  integer(4) :: plan
  integer(4) :: rank
  integer(8) :: n(*)
  integer(8) :: inembed(*), onembed(*)
  integer(8) :: istride, idist, ostride, odist
  type(cudaDataType) :: inputType, outputType, executionType
  integer(4) :: batch
  integer(8) :: workSize(*)
```

10.5.2. cufftXtQueryPlan

This function only supports multi-gpu 1D transforms. It returns a derived type, `factors`, which contains the number of strings, the decomposition of factors, and (in the case of power of 2 sizes) some other useful mask and shift elements, used in converting between permuted and linear indexes.

```
integer(4) function cufftXtQueryPlan(plan, factors, queryType)
  integer(4) :: plan
  type(cufftXt1DFactors) :: factors
  integer(4) :: queryType
```

10.5.3. cufftXtSetWorkAreaPolicy

This function overrides the work area associated with a plan. Currently, the `workAreaPolicy` can be specified as `CUFFT_WORKAREA_MINIMAL` and cuFFT will attempt to re-plan to use zero bytes of work area memory. See the CUFFT documentation for support of other features.

```
integer(4) function cufftXtSetWorkAreaPolicy(plan, workAreaPolicy, workSize)
  integer(4) :: plan
  integer(4) :: workAreaPolicy
  integer(8) :: workSize
```

10.5.4. cufftXtGetSizeMany

This function gives a more accurate estimate than cufftEstimateMany() of the size of the work area required, in bytes, given the specified plan parameters used for `cufftXtMakePlanMany` and taking into account any plan settings which may have been made.

```
integer(4) function cufftXtGetSizeMany(plan, rank, n, inembed, istride, &
    idist, inputType, onembed, ostride, odist, outputType, batch, workSize, &
    executionType)
  integer(4) :: plan
  integer(4) :: rank
  integer(8) :: n(*)
  integer(8) :: inembed(*), onembed(*)
  integer(8) :: istride, idist, ostride, odist
  type(cudaDataType) :: inputType, outputType, executionType
  integer(4) :: batch
  integer(8) :: workSize(*)
```

10.5.5. cufftXtSetWorkArea

This function overrides the work areas associated with a plan. If the work area was auto-allocated, cuFFT frees the auto-allocated space. The cufftExecute*() calls assume that the work area pointer is valid and that it points to a contiguous region in device memory that does not overlap with any other work area. If this is not the case, results are indeterminate.

```
integer(4) function cufftXtSetWorkArea(plan, workArea)
  integer(4) :: plan
  type(c_devptr) :: workArea(*)
```

10.5.6. cufftXtSetDistribution

This function registers and describes the data distribution for a subsequent FFT operation. The call to `cufftXtSetDistribution` must occur after the call to `cufftCreate` but before the call to `cufftMakePlan*`.

```
integer(4) function cufftXtSetDistribution( plan, boxIn, boxOut )
  integer(4) :: plan
  type(cufftBox3d) :: boxIn
  type(cufftBox3d) :: boxOut
```

10.6. CUFFTXT Execution Functions

This section contains the execution functions, which perform the actual Fourier transform, in the cufftXt library.

10.6.1. cufftXtExec

This function executes any Fourier transform regardless of precision and type. In case of complex-to-real and real-to-complex transforms, the `direction` argument is ignored. Otherwise, the transform direction is specified by the `direction` parameter. This function uses the GPU memory pointed to by `input` as input data, and stores the computed Fourier coefficients in the `output` array. If those are the same, this method does an in-place transform. Any valid data type for the `input` and `output` arrays are accepted.

```
integer(4) function cufftXtExec( plan, input, output, direction )
  integer :: plan
  real, dimension(*) :: input, output  ! Any data type is allowed
  integer :: direction
```

10.6.2. cufftXtExecDescriptor

This function executes any Fourier transform regardless of precision and type. In case of complex-to-real and real-to-complex transforms, the `direction` argument is ignored. Otherwise, the transform direction is specified by the `direction` parameter. This function stores the result in the specified output arrays.

```
integer(4) function cufftXtExecDescriptor( plan, input, output, direction )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
  integer :: direction
```

10.6.3. cufftXtExecDescriptorC2C

This function executes a single precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorC2C( plan, input, output, direction )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
  integer :: direction
```

10.6.4. cufftXtExecDescriptorZ2Z

This function executes a double precision complex-to-complex transform plan in the transform direction as specified by the direction parameter. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorZ2Z( plan, input, output, direction )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
  integer :: direction
```

10.6.5. cufftXtExecDescriptorR2C

This function executes a single precision real-to-complex transform plan. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorR2C( plan, input, output )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
```

10.6.6. cufftXtExecDescriptorD2Z

This function executes a double precision real-to-complex transform plan. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorD2Z( plan, input, output )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
```

10.6.7. cufftXtExecDescriptorC2R

This function executes a single precision complex-to-real transform plan. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorC2R( plan, input, output )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
```

10.6.8. cufftXtExecDescriptorZ2D

This function executes a double precision complex-to-real transform plan. This multiple GPU function currently supports in-place transforms only; the result will be stored in the input arrays.

```
integer(4) function cufftXtExecDescriptorZ2D( plan, input, output )
  integer :: plan
  type(cudaLibXtDesc) :: input, output
```

10.7. CUFFTXT Callback Functions

This section contains the cuFFTXt callback functions, which allow users to provide custom load and store callbacks for FFT operations.

10.7.1. cufftXtSetCallback

This function sets a callback routine of the specified type for the specified plan. The `callback_routine` argument is an array of device function pointers. The `caller_info` argument is an array of device pointers to user-specified data.

```
integer(4) function cufftXtSetCallback( plan, callback_routine, cbType, caller_info )
  integer(4) :: plan
  type(c_ptr) :: callback_routine(*)
  integer(4) :: cbType   ! cufftXtCallbackType value
  type(c_ptr) :: caller_info(*)
```

10.7.2. cufftXtClearCallback

This function clears a previously set callback routine of the specified type for the specified plan.

```
integer(4) function cufftXtClearCallback( plan, cbType )
  integer(4) :: plan
  integer(4) :: cbType   ! cufftXtCallbackType value
```

10.7.3. cufftXtSetCallbackSharedSize

This function sets the amount of shared memory, in bytes, that a callback can use for the specified plan and callback type.

```
integer(4) function cufftXtSetCallbackSharedSize( plan, cbType, sharedSize )
  integer(4) :: plan
  integer(4) :: cbType      ! cufftXtCallbackType value
  integer(8) :: sharedSize
```

10.7.4. cufftXtSetJITCallback

This function sets a JIT LTO callback for the specified plan and callback type. The `lto_callback_fatbin` is a pointer to the LTO-compiled fatbinary containing the callback, and `lto_callback_fatbin_size` is the size in bytes of that fatbinary.

```
integer(4) function cufftXtSetJITCallback( plan, lto_callback_symbol_name, &
       lto_callback_fatbin, lto_callback_fatbin_size, cbType, caller_info )
  integer(4) :: plan
  character(*) :: lto_callback_symbol_name
  type(c_ptr) :: lto_callback_fatbin
  integer(8) :: lto_callback_fatbin_size
  integer(4) :: cbType   ! cufftXtCallbackType value
  type(c_ptr) :: caller_info(*)
```

10.8. CUFFTMP Functions

This section contains the cuFFTMp functions which extend the cuFFTXt library functionality to multiple processes and multiple GPUs.

10.8.1. cufftMpAttachComm

This function attaches a communicator, such as MPI_COMM_WORLD, to a cuFFT plan, for later application of a distributed FFT operation

```
integer(4) function cufftMpAttachComm( plan, commType, fcomm )
  integer(4) :: plan
  integer(4) :: commType
  integer(4) :: fcomm
```

10.8.2. cufftMpCreateReshape

This function creates a cuFFTMp reshape handle for later application of a distributed FFT operation

```
integer(4) function cufftMpCreateReshape( reshapeHandle )
  type(c_ptr) :: reshapeHandle
```

10.8.3. cufftMpAttachReshapeComm

This function is deprecated. Use `cufftMpMakeReshape` with the `fcomm` and `commType` arguments instead.

This function attaches a communicator, such as MPI_COMM_WORLD, to a cuFFTMp reshape handle, for later application of a distributed FFT operation.

```
integer(4) function cufftMpAttachReshapeComm( reshapeHandle, commType, fcomm )
  type(c_ptr) :: reshapeHandle
  integer(4) :: commType
  integer(4) :: fcomm
```

10.8.4. cufftMpGetReshapeSize

This function returns the size needed for work space in the subsequent cuFFTMp reshape execution. Currently, a work area is not required, but that may change in future releases.

```
integer(4) function cufftMpGetReshapeSize( reshapeHandle, workSize )
   type(c_ptr) :: reshapeHandle
   integer(8)  :: workSize
```

10.8.5. cufftMpMakeReshape

This function creates a cuFFTMp reshape plan based on the input and output data distributions. Note that the lower, upper, and strides arrays use C conventions for bounds and strides. The `fcomm` and `commType` arguments specify the MPI communicator and communication type, replacing the need for a separate call to `cufftMpAttachReshapeComm`.

```
integer(4) function cufftMpMakeReshape( reshapeHandle, &
       elementSize, rank, &
       lower_input, upper_input, strides_input, &
       lower_output, upper_output, strides_output, &
       fcomm, commType )
   type(c_ptr) :: reshapeHandle
   integer(8)  :: elementSize
   integer(4)  :: rank
   integer(8)  :: lower_input(rank)
   integer(8)  :: upper_input(rank)
   integer(8)  :: strides_input(rank)
   integer(8)  :: lower_output(rank)
   integer(8)  :: upper_output(rank)
   integer(8)  :: strides_output(rank)
   integer(4)  :: fcomm
   integer(4)  :: commType
```

10.8.6. cufftMpExecReshapeAsync

This function executes a cuFFTMp reshape plan on the specified stream.

```
integer(4) function cufftMpExecReshapeAsync( reshapeHandle, &
       dataOut, dataIn, workSpace, stream )
   type(c_ptr) :: reshapeHandle
   type(c_devptr) :: dataOut
   type(c_devptr) :: dataIn
   type(c_devptr) :: workSpace
   integer(kind=cuda_stream_kind) :: stream
```

10.8.7. cufftMpDestroyReshape

This function destroys a cuFFTMp reshape handle.

```
integer(4) function cufftMpDestroyReshape( reshapeHandle )
  type(c_ptr) :: reshapeHandle
```

Chapter 11. Random Number Runtime Library APIs

This section describes the Fortran interfaces to the CUDA cuRAND library. The cuRAND functionality is accessible from both host and device code. In the host library, all of the runtime API routines are integer functions that return an error code; they return a value of CURAND_STATUS_SUCCESS if the call was successful, or other cuRAND return status value if there was an error. The host library routines are meant to produce a series or array of random numbers. In the device library, the init routines are subroutines and the generator functions return the type of the value being generated. The device library routines are meant for producing a single value per thread per call.

Chapter 10 contains examples of accessing the cuRAND library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed in host code by adding the line

```
use curand
```

to your program unit.

Unless a specific kind is provided, the plain integer type implies integer(4) and the plain real type implies real(4).

11.1. CURAND Definitions and Helper Functions

This section contains definitions and data types used in the cuRAND library and interfaces to the cuRAND helper functions.

The curand module contains the following derived type definitions:

```
TYPE curandGenerator
  TYPE(C_PTR)  :: handle
END TYPE
```

The curand module contains the following enumerations:

```
! CURAND Status
enum, bind(c)
    enumerator :: CURAND_STATUS_SUCCESS                 = 0
    enumerator :: CURAND_STATUS_VERSION_MISMATCH        = 100
    enumerator :: CURAND_STATUS_NOT_INITIALIZED         = 101
    enumerator :: CURAND_STATUS_ALLOCATION_FAILED       = 102
    enumerator :: CURAND_STATUS_TYPE_ERROR              = 103
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CURAND_STATUS_OUT_OF_RANGE              = 104
    enumerator :: CURAND_STATUS_LENGTH_NOT_MULTIPLE       = 105
    enumerator :: CURAND_STATUS_DOUBLE_PRECISION_REQUIRED = 106
    enumerator :: CURAND_STATUS_LAUNCH_FAILURE            = 201
    enumerator :: CURAND_STATUS_PREEXISTING_FAILURE       = 202
    enumerator :: CURAND_STATUS_INITIALIZATION_FAILED     = 203
    enumerator :: CURAND_STATUS_ARCH_MISMATCH             = 204
    enumerator :: CURAND_STATUS_INTERNAL_ERROR            = 999
end enum
```

```
! CURAND Generator Types
enum, bind(c)
    enumerator :: CURAND_RNG_TEST                    = 0
    enumerator :: CURAND_RNG_PSEUDO_DEFAULT          = 100
    enumerator :: CURAND_RNG_PSEUDO_XORWOW           = 101
    enumerator :: CURAND_RNG_PSEUDO_MRG32K3A         = 121
    enumerator :: CURAND_RNG_PSEUDO_MTGP32           = 141
    enumerator :: CURAND_RNG_PSEUDO_MT19937          = 142
    enumerator :: CURAND_RNG_PSEUDO_PHILOX4_32_10    = 161
    enumerator :: CURAND_RNG_QUASI_DEFAULT           = 200
    enumerator :: CURAND_RNG_QUASI_SOBOL32           = 201
    enumerator :: CURAND_RNG_QUASI_SCRAMBLED_SOBOL32 = 202
    enumerator :: CURAND_RNG_QUASI_SOBOL64           = 203
    enumerator :: CURAND_RNG_QUASI_SCRAMBLED_SOBOL64 = 204
end enum
```

```
! CURAND Memory Ordering
enum, bind(c)
    enumerator :: CURAND_ORDERING_PSEUDO_BEST    = 100
    enumerator :: CURAND_ORDERING_PSEUDO_DEFAULT = 101
    enumerator :: CURAND_ORDERING_PSEUDO_SEEDED  = 102
    enumerator :: CURAND_ORDERING_QUASI_DEFAULT  = 201
end enum
```

```
! CURAND Direction Vectors
enum, bind(c)
    enumerator :: CURAND_DIRECTION_VECTORS_32_JOEKUO6           = 101
    enumerator :: CURAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 = 102
    enumerator :: CURAND_DIRECTION_VECTORS_64_JOEKUO6           = 103
    enumerator :: CURAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 = 104
end enum
```

```
! CURAND Methods
enum, bind(c)
    enumerator :: CURAND_CHOOSE_BEST    = 0
    enumerator :: CURAND_ITR            = 1
    enumerator :: CURAND_KNUTH          = 2
    enumerator :: CURAND_HITR           = 3
    enumerator :: CURAND_M1             = 4
    enumerator :: CURAND_M2             = 5
    enumerator :: CURAND_BINARY_SEARCH  = 6
    enumerator :: CURAND_DISCRETE_GAUSS = 7
    enumerator :: CURAND_REJECTION      = 8
    enumerator :: CURAND_DEVICE_API     = 9
    enumerator :: CURAND_FAST_REJECTION = 10
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CURAND_3RD            = 11
    enumerator :: CURAND_DEFINITION     = 12
    enumerator :: CURAND_POISSON        = 13
end enum
```

11.1.1. curandCreateGenerator

This function creates a new random number generator of type rng. See the beginning of this section for valid values of rng.

```
integer(4) function curandCreateGenerator(generator, rng)
  type(curandGenerator) :: generator
  integer :: rng
```

11.1.2. curandCreateGeneratorHost

This function creates a new host CPU random number generator of type rng. See the beginning of this section for valid values of rng.

```
integer(4) function curandCreateGeneratorHost(generator, rng)
  type(curandGenerator) :: generator
  integer :: rng
```

11.1.3. curandDestroyGenerator

This function destroys an existing random number generator.

```
integer(4) function curandDestroyGenerator(generator)
  type(curandGenerator) :: generator
```

11.1.4. curandGetVersion

This function returns the version number of the cuRAND library.

```
integer(4) function curandGetVersion(version)
  integer(4) :: version
```

11.1.5. curandSetStream

This function sets the current stream for the cuRAND kernel launches.

```
integer(4) function curandSetStream(generator, stream)
  type(curandGenerator) :: generator
  integer(kind=c_intptr_t) :: stream
```

11.1.6. curandSetPseudoRandomGeneratorSeed

This function sets the seed value of the pseudo-random number generator.

```
integer(4) function curandSetPseudoRandomGeneratorSeed(generator, seed)
  type(curandGenerator) :: generator
  integer(8) :: seed
```

11.1.7. curandSetGeneratorOffset

This function sets the absolute offset of the pseudo or quasirandom number generator.

```
integer(4) function curandSetGeneratorOffset(generator, offset)
  type(curandGenerator) :: generator
  integer(8) :: offset
```

11.1.8. curandSetGeneratorOrdering

This function sets the ordering of results of the pseudo or quasirandom number generator.

```
integer(4) function curandSetGeneratorOrdering(generator, order)
  type(curandGenerator) :: generator
  integer(4) :: order
```

11.1.9. curandSetQuasiRandomGeneratorDimensions

This function sets number of dimensions of the quasirandom number generator.

```
integer(4) function curandSetQuasiRandomGeneratorDimensions(generator, num)
  type(curandGenerator) :: generator
  integer(4) :: num
```

11.2. CURAND Generator Functions

This section contains interfaces for the cuRAND generator functions.

11.2.1. curandGenerate

This function generates 32-bit pseudo or quasirandom numbers.

```
integer(4) function curandGenerate(generator, array, num )
  type(curandGenerator) :: generator
  integer(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

11.2.2. curandGenerateLongLong

This function generates 64-bit integer quasirandom numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateLongLong(generator, array, num )
  type(curandGenerator) :: generator
  integer(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

11.2.3. curandGenerateUniform

This function generates 32-bit floating point uniformly distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateUniform(generator, array, num )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

11.2.4. curandGenerateUniformDouble

This function generates 64-bit floating point uniformly distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateUniformDouble(generator, array, num )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
```

11.2.5. curandGenerateNormal

This function generates 32-bit floating point normally distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateNormal(generator, array, num, mean, stddev )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(4) :: mean, stddev
```

11.2.6. curandGenerateNormalDouble

This function generates 64-bit floating point normally distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGenerateNormalDouble(generator, array, num, mean, stddev )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(8) :: mean, stddev
```

11.2.7. curandGeneratePoisson

This function generates Poisson-distributed random numbers. The function curandGenerate() has also been overloaded to accept these function arguments.

```
integer(4) function curandGeneratePoisson(generator, array, num, lambda )
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(8) :: lambda
```

11.2.8. curandGenerateSeeds

This function sets the starting state of the generator.

```
integer(4) function curandGenerateSeeds(generator)
  type(curandGenerator) :: generator
```

11.2.9. curandGenerateLogNormal

This function generates 32-bit floating point log-normally distributed random numbers.

```
integer(4) function curandGenerateLogNormal(generator, array, num, mean, stddev )
  type(curandGenerator) :: generator
  real(4), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(4) :: mean, stddev
```

11.2.10. curandGenerateLogNormalDouble

This function generates 64-bit floating point log-normally distributed random numbers.

```
integer(4) function curandGenerateLogNormalDouble(generator, array, num, mean, stddev
↪)
  type(curandGenerator) :: generator
  real(8), device :: array(*) ! Host or device depending on the generator
  integer(kind=c_intptr_t) :: num
  real(8) :: mean, stddev
```

11.3. CURAND Device Definitions and Functions

This section contains definitions and data types used in the cuRAND device library and interfaces to the cuRAND functions.

The curand device module contains the following derived type definitions:

```
TYPE curandStateXORWOW
    integer(4) :: d
    integer(4) :: v(5)
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStateXORWOW
```

```
TYPE curandStateMRG32k3a
    real(8)    :: s1(3)
    real(8)    :: s2(3)
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStateMRG32k3a
```

```
TYPE curandStateSobol32
    integer(4) :: d
    integer(4) :: x
    integer(4) :: c
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: direction_vectors(32)
END TYPE curandStateSobol32
```

```
TYPE curandStateScrambledSobol32
    integer(4) :: d
    integer(4) :: x
    integer(4) :: c
    integer(4) :: direction_vectors(32)
END TYPE curandStateScrambledSobol32
```

```
TYPE curandStateSobol64
    integer(8) :: d
    integer(8) :: x
    integer(8) :: c
    integer(8) :: direction_vectors(32)
END TYPE curandStateSobol64
```

```
TYPE curandStateScrambledSobol64
    integer(8) :: d
    integer(8) :: x
    integer(8) :: c
    integer(8) :: direction_vectors(32)
END TYPE curandStateScrambledSobol64
```

```
TYPE curandStateMtgp32
    integer(4) :: s(MTGP32_STATE_SIZE)
    integer(4) :: offset
    integer(4) :: pIdx
    integer(kind=int_ptr_kind()) :: k
    integer(4) :: precise_double_flag
END TYPE curandStateMtgp32
```

```
TYPE curandStatePhilox4_32_10
    integer(4) :: ctr
    integer(4) :: output
    integer(2) :: key
    integer(4) :: state
    integer(4) :: boxmuller_flag
    integer(4) :: boxmuller_flag_double
    real(4)    :: boxmuller_extra
    real(8)    :: boxmuller_extra_double
END TYPE curandStatePhilox4_32_10
```

11.3.1. curand_Init

This overloaded device subroutine initializes the state for the random number generator. These device subroutines are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.1.1 curandInitXORWOW

This function initializes the state for the XORWOW random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitXORWOW(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStateXORWOW) :: state
```

11.3.1.2 curandInitMRG32k3a

This function initializes the state for the MRG32k3a random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitMRG32k3a(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStateMRG32k3a) :: state
```

11.3.1.3 curandInitPhilox4_32_10

This function initializes the state for the Philox4_32_10 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitPhilox4_32_10(seed, sequence, offset, state)
  integer(8) :: seed
  integer(8) :: sequence
  integer(8) :: offset
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.1.4 curandInitSobol32

This function initializes the state for the Sobol32 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitSobol32(direction_vectors, offset, state)
  integer :: direction_vectors(*)
  integer(4) :: offset
  TYPE(curandStateSobol32) :: state
```

11.3.1.5 curandInitScrambledSobol32

This function initializes the state for the scrambled Sobol32 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitScrambledSobol32(direction_vectors, scramble, offset, state)
  integer :: direction_vectors(*)
  integer(4) :: scramble
  integer(4) :: offset
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.1.6 curandInitSobol64

This function initializes the state for the Sobol64 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitSobol64(direction_vectors, offset, state)
  integer :: direction_vectors(*)
  integer(8) :: offset
  TYPE(curandStateSobol64) :: state
```

11.3.1.7 curandInitScrambledSobol64

This function initializes the state for the scrambled Sobol64 random number generator. The function curand_init() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
subroutine curandInitScrambledSobol64(direction_vectors, scramble, offset, state)
  integer :: direction_vectors(*)
  integer(8) :: scramble
  integer(8) :: offset
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.2. curand

This overloaded device function returns 32 or 64 bits or random data based on the state argument. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.2.1 curandGetXORWOW

This function returns 32 bits of pseudorandomness from the XORWOW random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.2.2 curandGetMRG32k3a

This function returns 32 bits of pseudorandomness from the MRG32k3a random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.2.3 curandGetPhilox4_32_10

This function returns 32 bits of pseudorandomness from the Philox4_32_10 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.2.4 curandGetSobol32

This function returns 32 bits of quasirandomness from the Sobol32 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.2.5 curandGetScrambledSobol32

This function returns 32 bits of quasirandomness from the scrambled Sobol32 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.2.6 curandGetSobol64

This function returns 64 bits of quasirandomness from the Sobol64 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.2.7 curandGetScrambledSobol64

This function returns 64 bits of quasirandomness from the scrambled Sobol64 random number generator. The function curand() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
integer(4) function curandGetScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.3. Curand_Normal

This overloaded device function returns a 32-bit floating point normally distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.3.1 curandNormalXORWOW

This function returns a 32-bit floating point normally distributed random number from an XORWOW generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.3.2 curandNormalMRG32k3a

This function returns a 32-bit floating point normally distributed random number from an MRG32k3a generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.3.3 curandNormalPhilox4_32_10

This function returns a 32-bit floating point normally distributed random number from a Philox4_32_10 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.3.4 curandNormalSobol32

This function returns a 32-bit floating point normally distributed random number from an Sobol32 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.3.5 curandNormalScrambledSobol32

This function returns a 32-bit floating point normally distributed random number from an scrambled Sobol32 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.3.6 curandNormalSobol64

This function returns a 32-bit floating point normally distributed random number from an Sobol64 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.3.7 curandNormalScrambledSobol64

This function returns a 32-bit floating point normally distributed random number from an scrambled Sobol64 generator. The function curand_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandNormalScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.4. Curand_Normal_Double

This overloaded device function returns a 64-bit floating point normally distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.4.1 curandNormalDoubleXORWOW

This function returns a 64-bit floating point normally distributed random number from an XORWOW generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.4.2 curandNormalDoubleMRG32k3a

This function returns a 64-bit floating point normally distributed random number from an MRG32k3a generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.4.3 curandNormalDoublePhilox4_32_10

This function returns a 64-bit floating point normally distributed random number from a Philox4_32_10 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.4.4 curandNormalDoubleSobol32

This function returns a 64-bit floating point normally distributed random number from an Sobol32 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.4.5 curandNormalDoubleScrambledSobol32

This function returns a 64-bit floating point normally distributed random number from an scrambled Sobol32 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.4.6 curandNormalDoubleSobol64

This function returns a 64-bit floating point normally distributed random number from an Sobol64 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.4.7 curandNormalDoubleScrambledSobol64

This function returns a 64-bit floating point normally distributed random number from an scrambled Sobol64 generator. The function curand_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandNormalDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.5. Curand_Log_Normal

This overloaded device function returns a 32-bit floating point log-normally distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.5.1 curandLogNormalXORWOW

This function returns a 32-bit floating point log-normally distributed random number from an XORWOW generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.5.2 curandLogNormalMRG32k3a

This function returns a 32-bit floating point log-normally distributed random number from an MRG32k3a generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.5.3 curandLogNormalPhilox4_32_10

This function returns a 32-bit floating point log-normally distributed random number from a Philox4_32_10 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.5.4 curandLogNormalSobol32

This function returns a 32-bit floating point log-normally distributed random number from an Sobol32 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.5.5 curandLogNormalScrambledSobol32

This function returns a 32-bit floating point log-normally distributed random number from an scrambled Sobol32 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.5.6 curandLogNormalSobol64

This function returns a 32-bit floating point log-normally distributed random number from an Sobol64 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.5.7 curandLogNormalScrambledSobol64

This function returns a 32-bit floating point log-normally distributed random number from an scrambled Sobol64 generator. The function curand_log_normal() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandLogNormalScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.6. Curand_Log_Normal_Double

This overloaded device function returns a 64-bit floating point log-normally distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.6.1 curandLogNormalDoubleXORWOW

This function returns a 64-bit floating point log-normally distributed random number from an XORWOW generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.6.2 curandLogNormalDoubleMRG32k3a

This function returns a 64-bit floating point log-normally distributed random number from an MRG32k3a generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.6.3 curandLogNormalDoublePhilox4_32_10

This function returns a 64-bit floating point log-normally distributed random number from a Philox4_32_10 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.6.4 curandLogNormalDoubleSobol32

This function returns a 64-bit floating point log-normally distributed random number from an Sobol32 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.6.5 curandLogNormalDoubleScrambledSobol32

This function returns a 64-bit floating point log-normally distributed random number from an scrambled Sobol32 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.6.6 curandLogNormalDoubleSobol64

This function returns a 64-bit floating point log-normally distributed random number from an Sobol64 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.6.7 curandLogNormalDoubleScrambledSobol64

This function returns a 64-bit floating point log-normally distributed random number from an scrambled Sobol64 generator. The function curand_log_normal_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandLogNormalDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.7. Curand_Uniform

This overloaded device function returns a 32-bit floating point uniformly distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.7.1 curandUniformXORWOW

This function returns a 32-bit floating point uniformly distributed random number from an XORWOW generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.7.2 curandUniformMRG32k3a

This function returns a 32-bit floating point uniformly distributed random number from an MRG32k3a generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.7.3 curandUniformPhilox4_32_10

This function returns a 32-bit floating point uniformly distributed random number from a Philox4_32_10 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformPhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.7.4 curandUniformSobol32

This function returns a 32-bit floating point uniformly distributed random number from an Sobol32 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.7.5 curandUniformScrambledSobol32

This function returns a 32-bit floating point uniformly distributed random number from an scrambled Sobol32 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.7.6 curandUniformSobol64

This function returns a 32-bit floating point uniformly distributed random number from an Sobol64 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.7.7 curandUniformScrambledSobol64

This function returns a 32-bit floating point uniformly distributed random number from an scrambled Sobol64 generator. The function curand_uniform() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(4) function curandUniformScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

11.3.8. Curand_Uniform_Double

This overloaded device function returns a 64-bit floating point uniformly distributed random number. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

11.3.8.1 curandUniformDoubleXORWOW

This function returns a 64-bit floating point uniformly distributed random number from an XORWOW generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleXORWOW(state)
  TYPE(curandStateXORWOW) :: state
```

11.3.8.2 curandUniformDoubleMRG32k3a

This function returns a 64-bit floating point uniformly distributed random number from an MRG32k3a generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleMRG32k3a(state)
  TYPE(curandStateMRG32k3a) :: state
```

11.3.8.3 curandUniformDoublePhilox4_32_10

This function returns a 64-bit floating point uniformly distributed random number from a Philox4_32_10 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoublePhilox4_32_10(state)
  TYPE(curandStatePhilox4_32_10) :: state
```

11.3.8.4 curandUniformDoubleSobol32

This function returns a 64-bit floating point uniformly distributed random number from an Sobol32 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleSobol32(state)
  TYPE(curandStateSobol32) :: state
```

11.3.8.5 curandUniformDoubleScrambledSobol32

This function returns a 64-bit floating point uniformly distributed random number from an scrambled Sobol32 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleScrambledSobol32(state)
  TYPE(curandStateScrambledSobol32) :: state
```

11.3.8.6 curandUniformDoubleSobol64

This function returns a 64-bit floating point uniformly distributed random number from an Sobol64 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleSobol64(state)
  TYPE(curandStateSobol64) :: state
```

11.3.8.7 curandUniformDoubleScrambledSobol64

This function returns a 64-bit floating point uniformly distributed random number from an scrambled Sobol64 generator. The function curand_uniform_double() has also been overloaded to accept these function arguments, as in CUDA C++. Device Functions are declared `attributes(device)` in CUDA Fortran and `!$acc routine() seq` in OpenACC.

```
real(8) function curandUniformDoubleScrambledSobol64(state)
  TYPE(curandStateScrambledSobol64) :: state
```

Chapter 12. SPARSE Matrix Runtime Library APIs

This section describes the Fortran interfaces to the CUDA cuSPARSE library. The cuSPARSE functions are only accessible from host code. All of the runtime API routines are integer functions that return an error code; they return a value of CUSPARSE_STATUS_SUCCESS if the call was successful, or another cuSPARSE status return value if there was an error.

Chapter 10 contains examples of accessing the cuSPARSE library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line

```
use cusparse
```

to your program unit.

A number of the function interfaces listed in this chapter can take host or device scalar arguments. Those functions have an additional v2 interface, which does not implicitly manage the pointer mode for these calls. See section 1.6 for further discussion on the handling of pointer modes.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

12.1. CUSPARSE Definitions and Helper Functions

This section contains definitions and data types used in the cuSPARSE library and interfaces to the cuSPARSE helper functions.

The cuSPARSE module contains the following derived type definitions:

```
type cusparseHandle
  type(c_ptr) :: handle
end type cusparseHandle
```

```
type :: cusparseMatDescr
     type(c_ptr) :: descr
end type cusparseMatDescr
```

```
! This type was removed in CUDA 11.0
type cusparseSolveAnalysisInfo
```

(continues on next page)

(continued from previous page)

```
     type(c_ptr) :: info
end type cusparseSolveAnalysisInfo

! This type was removed in CUDA 11.0
type cusparseHybMat
     type(c_ptr) :: mat
end type cusparseHybMat

type cusparseCsrsv2Info
     type(c_ptr) :: info
end type cusparseCsrsv2Info

type cusparseCsric02Info
     type(c_ptr) :: info
end type cusparseCsric02Info

type cusparseCsrilu02Info
     type(c_ptr) :: info
end type cusparseCsrilu02Info

type cusparseBsrsv2Info
     type(c_ptr) :: info
end type cusparseBsrsv2Info

type cusparseBsric02Info
     type(c_ptr) :: info
end type cusparseBsric02Info

type cusparseBsrilu02Info
     type(c_ptr) :: info
end type cusparseBsrilu02Info

type cusparseBsrsm2Info
     type(c_ptr) :: info
end type cusparseBsrsm2Info

type cusparseCsrgemm2Info
     type(c_ptr) :: info
end type cusparseCsrgemm2Info

type cusparseColorInfo
     type(c_ptr) :: info
end type cusparseColorInfo

type cusparseCsru2csrInfo
     type(c_ptr) :: info
end type cusparseCsru2csrInfo

type cusparseSpVecDescr
     type(c_ptr) :: descr
end type cusparseSpVecDescr
```

```
type cusparseDnVecDescr
     type(c_ptr) :: descr
end type cusparseDnVecDescr
```

```
type cusparseSpMatDescr
     type(c_ptr) :: descr
end type cusparseSpMatDescr
```

```
type cusparseDnMatDescr
     type(c_ptr) :: descr
end type cusparseDnMatDescr
```

```
type cusparseSpSVDescr
     type(c_ptr) :: descr
end type cusparseSpSVDescr
```

```
type cusparseSpSMDescr
     type(c_ptr) :: descr
end type cusparseSpSMDescr
```

```
type cusparseSpGEMMDescr
     type(c_ptr) :: descr
end type cusparseSpGEMMDescr
```

The cuSPARSE module contains the following constants and enumerations:

```
! cuSPARSE Version Info
  integer, parameter :: CUSPARSE_VER_MAJOR = 12
  integer, parameter :: CUSPARSE_VER_MINOR = 1
  integer, parameter :: CUSPARSE_VER_PATCH = 2
  integer, parameter :: CUSPARSE_VER_BUILD = 129
  integer, parameter :: CUSPARSE_VERSION   = (CUSPARSE_VER_MAJOR * 1000 + &
                          CUSPARSE_VER_MINOR *  100 + CUSPARSE_VER_PATCH)
```

```
! cuSPARSE status return values
enum, bind(C) ! cusparseStatus_t
    enumerator :: CUSPARSE_STATUS_SUCCESS=0
    enumerator :: CUSPARSE_STATUS_NOT_INITIALIZED=1
    enumerator :: CUSPARSE_STATUS_ALLOC_FAILED=2
    enumerator :: CUSPARSE_STATUS_INVALID_VALUE=3
    enumerator :: CUSPARSE_STATUS_ARCH_MISMATCH=4
    enumerator :: CUSPARSE_STATUS_MAPPING_ERROR=5
    enumerator :: CUSPARSE_STATUS_EXECUTION_FAILED=6
    enumerator :: CUSPARSE_STATUS_INTERNAL_ERROR=7
    enumerator :: CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED=8
    enumerator :: CUSPARSE_STATUS_ZERO_PIVOT=9
    enumerator :: CUSPARSE_STATUS_NOT_SUPPORTED=10
    enumerator :: CUSPARSE_STATUS_INSUFFICIENT_RESOURCES=11
end enum
```

```
enum, bind(c) ! cusparsePointerMode_t
    enumerator :: CUSPARSE_POINTER_MODE_HOST = 0
    enumerator :: CUSPARSE_POINTER_MODE_DEVICE = 1
end enum
```

```
enum, bind(c) ! cusparseAction_t
    enumerator :: CUSPARSE_ACTION_SYMBOLIC = 0
    enumerator :: CUSPARSE_ACTION_NUMERIC = 1
end enum
```

```
enum, bind(C) ! cusparseMatrixType_t
    enumerator :: CUSPARSE_MATRIX_TYPE_GENERAL = 0
    enumerator :: CUSPARSE_MATRIX_TYPE_SYMMETRIC = 1
    enumerator :: CUSPARSE_MATRIX_TYPE_HERMITIAN = 2
    enumerator :: CUSPARSE_MATRIX_TYPE_TRIANGULAR = 3
end enum
```

```
enum, bind(C) ! cusparseFillMode_t
    enumerator :: CUSPARSE_FILL_MODE_LOWER = 0
    enumerator :: CUSPARSE_FILL_MODE_UPPER = 1
end enum
```

```
enum, bind(C) ! cusparseDiagType_t
    enumerator :: CUSPARSE_DIAG_TYPE_NON_UNIT = 0
    enumerator :: CUSPARSE_DIAG_TYPE_UNIT = 1
end enum
```

```
enum, bind(C) ! cusparseIndexBase_t
    enumerator :: CUSPARSE_INDEX_BASE_ZERO = 0
    enumerator :: CUSPARSE_INDEX_BASE_ONE = 1
end enum
```

```
enum, bind(C) ! cusparseOperation_t
    enumerator :: CUSPARSE_OPERATION_NON_TRANSPOSE = 0
    enumerator :: CUSPARSE_OPERATION_TRANSPOSE = 1
    enumerator :: CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE = 2
end enum
```

```
enum, bind(C) ! cusparseDirection_t
    enumerator :: CUSPARSE_DIRECTION_ROW = 0
    enumerator :: CUSPARSE_DIRECTION_COLUMN = 1
end enum
```

```
enum, bind(C) ! cusparseHybPartition_t
    enumerator :: CUSPARSE_HYB_PARTITION_AUTO = 0
    enumerator :: CUSPARSE_HYB_PARTITION_USER = 1
    enumerator :: CUSPARSE_HYB_PARTITION_MAX = 2
end enum
```

```
enum, bind(C) ! cusparseSolvePolicy_t
    enumerator :: CUSPARSE_SOLVE_POLICY_NO_LEVEL = 0
    enumerator :: CUSPARSE_SOLVE_POLICY_USE_LEVEL = 1
end enum
```

```
enum, bind(C) ! cusparseSideMode_t
    enumerator :: CUSPARSE_SIDE_LEFT  = 0
    enumerator :: CUSPARSE_SIDE_RIGHT = 1
end enum
```

```
enum, bind(C) ! cusparseColorAlg_t
    enumerator :: CUSPARSE_COLOR_ALG0 = 0
    enumerator :: CUSPARSE_COLOR_ALG1 = 1
end enum
```

```
enum, bind(C) ! cusparseAlgMode_t;
    enumerator :: CUSPARSE_ALG0           = 0
    enumerator :: CUSPARSE_ALG1           = 1
    enumerator :: CUSPARSE_ALG_NAIVE      = 0
    enumerator :: CUSPARSE_ALG_MERGE_PATH = 0
end enum
```

```
enum, bind(C) ! cusparseCsr2CscAlg_t;
    enumerator :: CUSPARSE_CSR2CSC_ALG_DEFAULT = 1
    enumerator :: CUSPARSE_CSR2CSC_ALG1 = 1
    enumerator :: CUSPARSE_CSR2CSC_ALG2 = 2
end enum
```

```
enum, bind(C) ! cusparseFormat_t;
    enumerator :: CUSPARSE_FORMAT_CSR     = 1
    enumerator :: CUSPARSE_FORMAT_CSC     = 2
    enumerator :: CUSPARSE_FORMAT_COO     = 3
    enumerator :: CUSPARSE_FORMAT_COO_AOS = 4
    enumerator :: CUSPARSE_FORMAT_BLOCKED_ELL = 5
    enumerator :: CUSPARSE_FORMAT_BSR     = 6
    enumerator :: CUSPARSE_FORMAT_SLICED_ELLPACK = 7
end enum
```

```
enum, bind(C) ! cusparseOrder_t;
    enumerator :: CUSPARSE_ORDER_COL = 1
    enumerator :: CUSPARSE_ORDER_ROW = 2
end enum
```

```
enum, bind(C) ! cusparseSpMVAlg_t;
    enumerator :: CUSPARSE_MV_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_COOMV_ALG      = 1
    enumerator :: CUSPARSE_CSRMV_ALG1     = 2
    enumerator :: CUSPARSE_CSRMV_ALG2     = 3
    enumerator :: CUSPARSE_SPMV_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_SPMV_CSR_ALG1    = 2
    enumerator :: CUSPARSE_SPMV_CSR_ALG2    = 3
    enumerator :: CUSPARSE_SPMV_COO_ALG1    = 1
    enumerator :: CUSPARSE_SPMV_COO_ALG2    = 4
    enumerator :: CUSPARSE_SPMV_SELL_ALG1   = 5
end enum
```

```
enum, bind(C) ! cusparseSpMMAlg_t;
    enumerator :: CUSPARSE_MM_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_COOMM_ALG1 = 1
    enumerator :: CUSPARSE_COOMM_ALG2 = 2
    enumerator :: CUSPARSE_COOMM_ALG3 = 3
    enumerator :: CUSPARSE_CSRMM_ALG1 = 4
    enumerator :: CUSPARSE_SPMM_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_SPMM_COO_ALG1    = 1
    enumerator :: CUSPARSE_SPMM_COO_ALG2    = 2
    enumerator :: CUSPARSE_SPMM_COO_ALG3    = 3
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CUSPARSE_SPMM_COO_ALG4    = 5
    enumerator :: CUSPARSE_SPMM_CSR_ALG1    = 4
    enumerator :: CUSPARSE_SPMM_CSR_ALG2    = 6
    enumerator :: CUSPARSE_SPMM_CSR_ALG3    = 12
    enumerator :: CUSPARSE_SPMM_BLOCKED_ELL_ALG1 = 13
end enum
```

```
enum, bind(C) ! cusparseIndexType_t;
    enumerator :: CUSPARSE_INDEX_16U = 1
    enumerator :: CUSPARSE_INDEX_32I = 2
    enumerator :: CUSPARSE_INDEX_64I = 3
end enum
```

```
enum, bind(C) ! cusparseSpMatAttribute_t;
    enumerator :: CUSPARSE_SPMAT_FILL_MODE = 0
    enumerator :: CUSPARSE_SPMAT_DIAG_TYPE = 1
end enum
```

```
enum, bind(C) ! cusparseSparseToDenseAlg_t;
    enumerator :: CUSPARSE_SPARSETODENSE_ALG_DEFAULT = 0
    enumerator :: CUSPARSE_DENSETOSPARSE_ALG_DEFAULT = 0
end enum
```

```
enum, bind(C) ! cusparseSpSVAlg_t;
    enumerator :: CUSPARSE_SPSV_ALG_DEFAULT = 0
end enum
```

```
enum, bind(C) ! cusparseSpSVUpdate_t;
    enumerator :: CUSPARSE_SPSV_UPDATE_GENERAL  = 0
    enumerator :: CUSPARSE_SPSV_UPDATE_DIAGONAL = 1
end enum
```

```
enum, bind(C) ! cusparseSpSMAlg_t;
    enumerator :: CUSPARSE_SPSM_ALG_DEFAULT = 0
end enum
```

```
enum, bind(C) ! cusparseSpMMOpAlg_t;
    enumerator :: CUSPARSE_SPMM_OP_ALG_DEFAULT = 0
end enum
```

```
enum, bind(C) ! cusparseSpGEMMAlg_t;
    enumerator :: CUSPARSE_SPGEMM_DEFAULT = 0
    enumerator :: CUSPARSE_SPGEMM_CSR_ALG_DETERMINISTIC = 1
    enumerator :: CUSPARSE_SPGEMM_CSR_ALG_DETERMINITIC  = 1
    enumerator :: CUSPARSE_SPGEMM_CSR_ALG_NONDETERMINISTIC = 2
    enumerator :: CUSPARSE_SPGEMM_CSR_ALG_NONDETERMINITIC  = 2
    enumerator :: CUSPARSE_SPGEMM_ALG1 = 3
    enumerator :: CUSPARSE_SPGEMM_ALG2 = 4
    enumerator :: CUSPARSE_SPGEMM_ALG3 = 5
end enum
```

```
enum, bind(C) ! cusparseSDDMMAlg_t;
    enumerator :: CUSPARSE_SDDMM_ALG_DEFAULT = 0
end enum
```

12.1.1. cusparseCreate

This function initializes the cuSPARSE library and creates a handle on the cuSPARSE context. It must be called before any other cuSPARSE API function is invoked. It allocates hardware resources necessary for accessing the GPU.

```
integer(4) function cusparseCreate(handle)
  type(cusparseHandle) :: handle
```

12.1.2. cusparseDestroy

This function releases CPU-side resources used by the cuSPARSE library. The release of GPU-side resources may be deferred until the application shuts down.

```
integer(4) function cusparseDestroy(handle)
  type(cusparseHandle) :: handle
```

12.1.3. cusparseGetErrorName

This function returns the error code name.

```
character*128 function cusparseGetErrorName(ierr)
   integer(c_int) :: ierr
```

12.1.4. cusparseGetErrorString

This function returns the description string for an error code.

```
character*128 function cusparseGetErrorString(ierr)
   integer(c_int) :: ierr
```

12.1.5. cusparseGetVersion

This function returns the version number of the cuSPARSE library.

```
integer(4) function cusparseGetVersion(handle, version)
  type(cusparseHandle) :: handle
  integer(c_int) :: version
```

12.1.6. cusparseSetStream

This function sets the stream to be used by the cuSPARSE library to execute its routines.

```
integer(4) function cusparseSetStream(handle, stream)
  type(cusparseHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

12.1.7. cusparseGetStream

This function gets the stream used by the cuSPARSE library to execute its routines. If the cuSPARSE library stream is not set, all kernels use the default NULL stream.

```
integer(4) function cusparseGetStream(handle, stream)
  type(cusparseHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

12.1.8. cusparseGetPointerMode

This function obtains the pointer mode used by the cuSPARSE library. Please see section 1.6 for more details on pointer modes.

```
integer(4) function cusparseGetPointerMode(handle, mode)
  type(cusparseHandle) :: handle
  integer(c_int) :: mode
```

12.1.9. cusparseSetPointerMode

This function sets the pointer mode used by the cuSPARSE library. In these Fortran interfaces, this only has an effect when using the *_v2 interfaces. The default is for the values to be passed by reference on the host. Please see section 1.6 for more details on pointer modes.

```
integer(4) function cusparseSetPointerMode(handle, mode)
  type(cusparseHandle) :: handle
  integer(4) :: mode
```

12.1.10. cusparseCreateMatDescr

This function initializes the matrix descriptor. It sets the fields MatrixType and IndexBase to the default values CUSPARSE_MATRIX_TYPE_GENERAL and CUSPARSE_INDEX_BASE_ZERO , respectively, while leaving other fields uninitialized.

```
integer(4) function cusparseCreateMatDescr(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.11. cusparseDestroyMatDescr

This function releases the memory allocated for the matrix descriptor.

```
integer(4) function cusparseDestroyMatDescr(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.12. cusparseSetMatType

This function sets the MatrixType of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

12.1.13. cusparseGetMatType

This function returns the MatrixType of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatType(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.14. cusparseSetMatFillMode

This function sets the FillMode field of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatFillMode(descrA, mode)
  type(cusparseMatDescr) :: descrA
  integer(4) :: mode
```

12.1.15. cusparseGetMatFillMode

This function returns the FillMode field of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatFillMode(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.16. cusparseSetMatDiagType

This function sets the DiagType of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatDiagType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

12.1.17. cusparseGetMatDiagType

This function returns the DiagType of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatDiagType(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.18. cusparseSetMatIndexBase

This function sets the IndexBase field of the matrix descriptor descrA.

```
integer(4) function cusparseSetMatIndexBase(descrA, base)
  type(cusparseMatDescr) :: descrA
  integer(4) :: base
```

12.1.19. cusparseGetMatIndexBase

This function returns the IndexBase field of the matrix descriptor descrA.

```
integer(4) function cusparseGetMatIndexBase(descrA)
  type(cusparseMatDescr) :: descrA
```

12.1.20. cusparseCreateSolveAnalysisInfo

This function creates and initializes the solve and analysis structure to default values. This function, and all functions which use the cusparseSolveAnalysisInfo type, were removed in CUDA 11.0.

```
integer(4) function cusparseCreateSolveAnalysisInfo(info)
  type(cusparseSolveAnalysisinfo) :: info
```

12.1.21. cusparseDestroySolveAnalysisInfo

This function destroys and releases any memory required by the structure. This function, and all functions which use the cusparseSolveAnalysisInfo type, were removed in CUDA 11.0.

```
integer(4) function cusparseDestroySolveAnalysisInfo(info)
  type(cusparseSolveAnalysisinfo) :: info
```

12.1.22. cusparseGetLevelInfo

This function returns the number of levels and the assignment of rows into the levels computed by either the csrsv_analysis, csrsm_analysis or hybsv_analysis routines.

```
integer(4) function cusparseGetLevelInfo(handle, info, nlevels, levelPtr, levelInd)
  type(cusparseHandle) :: handle
  type(cusparseSolveAnalysisinfo) :: info
  integer(c_int) :: nlevels
  type(c_ptr) :: levelPtr
  type(c_ptr) :: levelInd
```

12.1.23. cusparseCreateHybMat

This function creates and initializes the hybA opaque data structure. This function, and all functions which use the cusparseHybMat type, were removed in CUDA 11.0.

```
integer(4) function cusparseCreateHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

12.1.24. cusparseDestroyHybMat

This function destroys and releases any memory required by the hybA structure. This function, and all functions which use the cusparseHybMat type, were removed in CUDA 11.0.

```
integer(4) function cusparseDestroyHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

12.1.25. cusparseCreateCsrsv2Info

This function creates and initializes the solve and analysis structure of csrsv2 to default values.

```
integer(4) function cusparseCreateCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

12.1.26. cusparseDestroyCsrsv2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

12.1.27. cusparseCreateCsric02Info

This function creates and initializes the solve and analysis structure of incomplete Cholesky to default values.

```
integer(4) function cusparseCreateCsric02Info(info)
  type(cusparseCsric02Info) :: info
```

12.1.28. cusparseDestroyCsric02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsric02Info(info)
  type(cusparseCsric02Info) :: info
```

12.1.29. cusparseCreateCsrilu02Info

This function creates and initializes the solve and analysis structure of incomplete LU to default values.

```
integer(4) function cusparseCreateCsrilu02Info(info)
  type(cusparseCsrilu02Info) :: info
```

12.1.30. cusparseDestroyCsrilu02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrilu02Info(info)
  type(cusparseCsrilu02Info) :: info
```

12.1.31. cusparseCreateBsrsv2Info

This function creates and initializes the solve and analysis structure of bsrsv2 to default values.

```
integer(4) function cusparseCreateBsrsv2Info(info)
  type(cusparseBsrsv2Info) :: info
```

12.1.32. cusparseDestroyBsrsv2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsv2Info(info)
  type(cusparseBsrsv2Info) :: info
```

12.1.33. cusparseCreateBsric02Info

This function creates and initializes the solve and analysis structure of block incomplete Cholesky to default values.

```
integer(4) function cusparseCreateBsric02Info(info)
  type(cusparseBsric02Info) :: info
```

12.1.34. cusparseDestroyBsric02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsric02Info(info)
  type(cusparseBsric02Info) :: info
```

12.1.35. cusparseCreateBsrilu02Info

This function creates and initializes the solve and analysis structure of block incomplete LU to default values.

```
integer(4) function cusparseCreateBsrilu02Info(info)
  type(cusparseBsrilu02Info) :: info
```

12.1.36. cusparseDestroyBsrilu02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrilu02Info(info)
  type(cusparseBsrilu02Info) :: info
```

12.1.37. cusparseCreateBsrsm2Info

This function creates and initializes the solve and analysis structure of bsrsm2 to default values.

```
integer(4) function cusparseCreateBsrsm2Info(info)
  type(cusparseBsrsm2Info) :: info
```

12.1.38. cusparseDestroyBsrsm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsm2Info(info)
  type(cusparseBsrsm2Info) :: info
```

12.1.39. cusparseCreateCsrgemm2Info

This function creates and initializes the analysis structure of general sparse matrix-matrix multiplication.

```
integer(4) function cusparseCreateCsrgemm2Info(info)
  type(cusparseCsrgemm2Info) :: info
```

12.1.40. cusparseDestroyCsrgemm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrgemm2Info(info)
  type(cusparseCsrgemm2Info) :: info
```

12.1.41. cusparseCreateColorInfo

This function creates coloring information used in calls like CSRCOLOR.

```
integer(4) function cusparseCreateColorInfo(info)
  type(cusparseColorInfo) :: info
```

12.1.42. cusparseDestroyColorInfo

This function destroys coloring information used in calls like CSRCOLOR.

```
integer(4) function cusparseDestroyColorInfo(info)
  type(cusparseColorInfo) :: info
```

12.1.43. cusparseCreateCsru2csrInfo

This function creates sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseCreateCsru2csrInfo(info)
  type(cusparseCsru2csrInfo) :: info
```

12.1.44. cusparseDestroyCsru2csrInfo

This function creates sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseDestroyCsru2csrInfo(info)
  type(cusparseCsru2csrInfo) :: info
```

12.2. CUSPARSE Level 1 Functions

This section contains interfaces for the level 1 sparse linear algebra functions that perform operations between dense and sparse vectors.

12.2.1. cusparseSaxpyi

SAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseSaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: alpha ! device or host variable
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  integer :: idxBase
```

12.2.2. cusparseDaxpyi

DAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseDaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: alpha ! device or host variable
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  integer :: idxBase
```

12.2.3. cusparseCaxpyi

CAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseCaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(4), device :: alpha ! device or host variable
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  integer :: idxBase
```

12.2.4. cusparseZaxpyi

ZAXPY performs constant times a vector plus a vector. This function multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format, i.e. y = y + alpha * xVal(xInd)

```
integer(4) function cusparseZaxpyi(handle, nnz, alpha, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
```

(continues on next page)

(continued from previous page)

```
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer :: idxBase
```

12.2.5. cusparseSdoti

SDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseSdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  real(4), device :: res ! device or host variable
  integer :: idxBase
```

12.2.6. cusparseDdoti

DDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseDdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  real(8), device :: res ! device or host variable
  integer :: idxBase
```

12.2.7. cusparseCdoti

CDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseCdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  complex(4), device :: res ! device or host variable
  integer :: idxBase
```

12.2.8. cusparseZdoti

ZDOT forms the dot product of two vectors. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseZdoti(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  complex(8), device :: res ! device or host variable
  integer :: idxBase
```

12.2.9. cusparseCdotci

CDOTC forms the dot product of two vectors, conjugating the first vector. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * conjg(xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseCdotci(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  complex(4), device :: res ! device or host variable
  integer :: idxBase
```

12.2.10. cusparseZdotci

ZDOTC forms the dot product of two vectors, conjugating the first vector. This function returns the dot product of a vector x in sparse format and vector y in dense format, i.e. res = sum(y * conjg(xVal(xInd))

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpVV

```
integer(4) function cusparseZdotci(handle, nnz, xVal, xInd, y, res, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  complex(8), device :: res ! device or host variable
  integer :: idxBase
```

12.2.11. cusparseSgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(4), device :: y(*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.12. cusparseDgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: y(*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.13. cusparseCgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: y(*)
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.14. cusparseZgthr

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd) Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: y(*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.15. cusparseSgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(4), device :: y(*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.16. cusparseDgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: y(*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.17. cusparseCgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: y(*)
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.18. cusparseZgthrz

This function gathers the elements of the vector y listed in the index array xInd into data array xVal, i.e. xVal = y(xInd); y(xInd) = 0.0 Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: y(*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase
```

12.2.19. cusparseSsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseSsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  integer(4) :: idxBase
```

12.2.20. cusparseDsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseDsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  integer(4) :: idxBase
```

12.2.21. cusparseCsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseCsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  integer(4) :: idxBase
```

12.2.22. cusparseZsctr

This function scatters the elements of the dense format vector x into the vector y in sparse format, i.e. y(xInd) = x Fortran programmers should normally use idxBase == 1.

```
integer(4) function cusparseZsctr(handle, nnz, xVal, xInd, y, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer(4) :: idxBase
```

12.2.23. cusparseSroti

SROT applies a plane rotation. X is a sparse vector and Y is dense.

```
integer(4) function cusparseSroti(handle, nnz, xVal, xInd, y, c, s, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  real(4), device :: c, s ! device or host variable
  integer :: idxBase
```

12.2.24. cusparseDroti

DROT applies a plane rotation. X is a sparse vector and Y is dense.

```
integer(4) function cusparseDroti(handle, nnz, xVal, xInd, y, c, s, idxBase)
  type(cusparseHandle) :: handle
  integer :: nnz
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
```

(continues on next page)

(continued from previous page)

```
  real(8), device :: y(*)
  real(8), device :: c, s ! device or host variable
  integer :: idxBase
```

12.3. CUSPARSE Level 2 Functions

This section contains interfaces for the level 2 sparse linear algebra functions that perform operations between sparse matrices and dense vectors.

12.3.1. cusparseSbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseSbsrmv(handle, dir, trans, mb, nb, nnzb, alpha, descr,
↪bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

12.3.2. cusparseDbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseDbsrmv(handle, dir, trans, mb, nb, nnzb, alpha, descr,
↪bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  real(8), device :: alpha ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  type(cusparseMatDescr) :: descr
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

12.3.3. cusparseCbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseCbsrmv(handle, dir, trans, mb, nb, nnzb, alpha, descr,
↪bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

12.3.4. cusparseZbsrmv

BSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an (mb*blockDim) x (nb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrVal, bsrRowPtr, and bsrColInd

```
integer(4) function cusparseZbsrmv(handle, dir, trans, mb, nb, nnzb, alpha, &
           descr, bsrVal, bsrRowPtr, bsrColInd, blockDim, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: mb, nb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*)
  integer(4), device :: bsrColInd(*)
```

(continues on next page)

(continued from previous page)

```
  integer :: blockDim
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

12.3.5. cusparseSbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseSbsrxmv(handle, dir, trans, sizeOfMask, mb, nb, nnzb,
↪alpha, &
           descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd, blockDim, x,
↪beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

12.3.6. cusparseDbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseDbsrxmv(handle, dir, trans, sizeOfMask, mb, nb, nnzb,
↪alpha, &
           descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd, blockDim, x,
↪beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

12.3.7. cusparseCbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseCbsrxmv(handle, dir, trans, sizeOfMask, mb, nb, nnzb,
↪alpha, &
           descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd, blockDim, x,
↪beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

12.3.8. cusparseZbsrxmv

BSRXMV performs a BSRMV and a mask operation.

```
integer(4) function cusparseZbsrxmv(handle, dir, trans, sizeOfMask, mb, nb, nnzb,
↪alpha, &
           descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd, blockDim, x,
↪beta, y)
  type(cusparseHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrMaskPtr(*), bsrRowPtr(*), bsrEndPtr(*)
  integer(4), device :: bsrColInd(*)
  integer :: blockDim
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

12.3.9. cusparseScsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMV

```
integer(4) function cusparseScsrmv(handle, trans, m, n, nnz, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

12.3.10. cusparseDcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMV

```
integer(4) function cusparseDcsrmv(handle, trans, m, n, nnz, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

12.3.11. cusparseCcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMV

```
integer(4) function cusparseCcsrmv(handle, trans, m, n, nnz, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

12.3.12. cusparseZcsrmv

CSRMV performs one of the matrix-vector operations y := alpha*A*x + beta*y, or y := alpha*A**T*x + beta*y, where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMV

```
integer(4) function cusparseZcsrmv(handle, trans, m, n, nnz, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

12.3.13. cusparseScsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseScsrsv_analysis(handle, trans, m, nnz, descr, csrVal,
↪csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.14. cusparseDcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseDcsrsv_analysis(handle, trans, m, nnz, descr, csrVal,
↪csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.15. cusparseCcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseCcsrsv_analysis(handle, trans, m, nnz, descr, csrVal,
↪csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.16. cusparseZcsrsv_analysis

This function performs the analysis phase of csrsv.

```
integer(4) function cusparseZcsrsv_analysis(handle, trans, m, nnz, descr, csrVal,
↪csrRowPtr, csrColInd, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.17. cusparseScsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseScsrsv_solve(handle, trans, m, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

12.3.18. cusparseDcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseDcsrsv_solve(handle, trans, m, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(8), device :: x(*)
  real(8), device :: y(*)
```

12.3.19. cusparseCcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseCcsrsv_solve(handle, trans, m, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

12.3.20. cusparseZcsrsv_solve

This function performs the solve phase of csrsv.

```
integer(4) function cusparseZcsrsv_solve(handle, trans, m, alpha, descr, csrVal,
↪csrRowPtr, csrColInd, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  integer :: m
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: x(*)
  complex(8), device :: y(*)
```

12.3.21. cusparseSgemvi_bufferSize

This function returns the buffer size, in bytes, needed by cusparseSgemvi.

```
integer(4) function cusparseSgemvi_bufferSize(handle, transA, m, n, nnz, pBufferSize)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize
```

12.3.22. cusparseDgemvi_bufferSize

This function returns the buffer size, in bytes, needed by cusparseDgemvi.

```
integer(4) function cusparseDgemvi_bufferSize(handle, transA, m, n, nnz, pBufferSize)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize
```

12.3.23. cusparseCgemvi_bufferSize

This function returns the buffer size, in bytes, needed by cusparseCgemvi.

```
integer(4) function cusparseCgemvi_bufferSize(handle, transA, m, n, nnz, pBufferSize)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize
```

12.3.24. cusparseZgemvi_bufferSize

This function returns the buffer size, in bytes, needed by cusparseZgemvi.

```
integer(4) function cusparseZgemvi_bufferSize(handle, transA, m, n, nnz, pBufferSize)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize
```

12.3.25. cusparseSgemvi

GEMVI performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, A is an m x n dense matrix, x is a sparse vector, and y is a dense vector.

```
integer(4) function cusparseSgemvi(handle, transA, m, n, alpha, A, lda, nnz, xVal,
↪xInd, beta, y, idxBase, pBuffer)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, lda, nnz, idxBase
  real(4), device :: alpha, beta ! device or host variable
  real(4), device :: A(lda,*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(4), device :: y(*)
  integer(1), device :: pBuffer(*) ! Any data type is allowed
```

12.3.26. cusparseDgemvi

GEMVI performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, A is an m x n dense matrix, x is a sparse vector, and y is a dense vector.

```
integer(4) function cusparseDgemvi(handle, transA, m, n, alpha, A, lda, nnz, xVal,
↪xInd, beta, y, idxBase, pBuffer)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, lda, nnz, idxBase
  real(8), device :: alpha, beta ! device or host variable
  real(8), device :: A(lda,*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  real(8), device :: y(*)
  integer(1), device :: pBuffer(*) ! Any data type is allowed
```

12.3.27. cusparseCgemvi

GEMVI performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, A is an m x n dense matrix, x is a sparse vector, and y is a dense vector.

```
integer(4) function cusparseCgemvi(handle, transA, m, n, alpha, A, lda, nnz, xVal,
↪xInd, beta, y, idxBase, pBuffer)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, lda, nnz, idxBase
  complex(4), device :: alpha, beta ! device or host variable
  complex(4), device :: A(lda,*)
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  integer(1), device :: pBuffer(*) ! Any data type is allowed
```

12.3.28. cusparseZgemvi

GEMVI performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, A is an m x n dense matrix, x is a sparse vector, and y is a dense vector.

```
integer(4) function cusparseZgemvi(handle, transA, m, n, alpha, A, lda, nnz, xVal,
↪xInd, beta, y, idxBase, pBuffer)
  type(cusparseHandle) :: handle
  integer :: transA
  integer :: m, n, lda, nnz, idxBase
  complex(8), device :: alpha, beta ! device or host variable
  complex(8), device :: A(lda,*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer(1), device :: pBuffer(*) ! Any data type is allowed
```

12.3.29. cusparseShybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

This function was removed in CUDA 11.0.

```
integer(4) function cusparseShybmv(handle, trans, alpha, descr, hyb, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  real(4), device :: x(*)
  real(4), device :: beta ! device or host variable
  real(4), device :: y(*)
```

12.3.30. cusparseDhybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

This function was removed in CUDA 11.0.

```
integer(4) function cusparseDhybmv(handle, trans, alpha, descr, hyb, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  real(8), device :: x(*)
  real(8), device :: beta ! device or host variable
  real(8), device :: y(*)
```

12.3.31. cusparseChybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

This function was removed in CUDA 11.0.

```
integer(4) function cusparseChybmv(handle, trans, alpha, descr, hyb, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  complex(4), device :: x(*)
  complex(4), device :: beta ! device or host variable
  complex(4), device :: y(*)
```

12.3.32. cusparseZhybmv

HYBMV performs the matrix-vector operations y := alpha*A*x + beta*y where alpha and beta are scalars, x and y are vectors and A is an m x n sparse matrix that is defined in the HYB storage format.

This function was removed in CUDA 11.0.

```
integer(4) function cusparseZhybmv(handle, trans, alpha, descr, hyb, x, beta, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  complex(8), device :: x(*)
  complex(8), device :: beta ! device or host variable
  complex(8), device :: y(*)
```

12.3.33. cusparseShybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseShybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.34. cusparseDhybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseDhybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.35. cusparseChybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseChybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.36. cusparseZhybsv_analysis

This function performs the analysis phase of hybsv.

```
integer(4) function cusparseZhybsv_analysis(handle, trans, descr, hyb, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
```

12.3.37. cusparseShybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseShybsv_solve(handle, trans, alpha, descr, hyb, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

12.3.38. cusparseDhybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseDhybsv_solve(handle, trans, alpha, descr, hyb, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  real(8), device :: x(*)
  real(8), device :: y(*)
```

12.3.39. cusparseChybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseChybsv_solve(handle, trans, alpha, descr, hyb, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
```

(continues on next page)

(continued from previous page)

```
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

12.3.40. cusparseZhybsv_solve

This function performs the solve phase of hybsv.

```
integer(4) function cusparseZhybsv_solve(handle, trans, alpha, descr, hyb, info, x, y)
  type(cusparseHandle) :: handle
  integer :: trans
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descr
  type(cusparseHybMat) :: hyb
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: x(*)
  complex(8), device :: y(*)
```

12.3.41. cusparseSbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseSbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.42. cusparseDbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseDbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
```

(continues on next page)

(continued from previous page)

```
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.43. cusparseCbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseCbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.44. cusparseZbsrsv2_bufferSize

This function returns the size of the buffer used in bsrsv2.

```
integer(4) function cusparseZbsrsv2_bufferSize(handle, dirA, transA, mb, nnzb, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.45. cusparseSbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseSbsrsv2_analysis(handle, dirA, transA, mb, nnzb, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
```

(continues on next page)

(continued from previous page)

```
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.46. cusparseDbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseDbsrsv2_analysis(handle, dirA, transA, mb, nnzb, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.47. cusparseCbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseCbsrsv2_analysis(handle, dirA, transA, mb, nnzb, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.48. cusparseZbsrsv2_analysis

This function performs the analysis phase of bsrsv2.

```
integer(4) function cusparseZbsrsv2_analysis(handle, dirA, transA, mb, nnzb, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
```

(continues on next page)

(continued from previous page)

```
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.49. cusparseSbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseSbsrsv2_solve(handle, dirA, transA, mb, nnzb, &
          alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  real(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.50. cusparseDbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseDbsrsv2_solve(handle, dirA, transA, mb, nnzb, &
          alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  real(8), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.51. cusparseCbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseCbsrsv2_solve(handle, dirA, transA, mb, nnzb, alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  complex(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.52. cusparseZbsrsv2_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparseZbsrsv2_solve(handle, dirA, transA, mb, nnzb, alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparseBsrsv2Info) :: info
  complex(8), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.53. cusparseXbsrsv2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrsv2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrsv2Info) :: info
  integer(4), device :: position ! device or host variable
```

12.3.54. cusparseScsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseScsrsv2_bufferSize(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.55. cusparseDcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseDcsrsv2_bufferSize(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.56. cusparseCcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseCcsrsv2_bufferSize(handle, transA, m, nnz, descrA,
↪csrValA, &
           csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.57. cusparseZcsrsv2_bufferSize

This function returns the size of the buffer used in csrsv2.

```
integer(4) function cusparseZcsrsv2_bufferSize(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.3.58. cusparseScsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseScsrsv2_analysis(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.59. cusparseDcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseDcsrsv2_analysis(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.60. cusparseCcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseCcsrsv2_analysis(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.61. cusparseZcsrsv2_analysis

This function performs the analysis phase of csrsv2.

```
integer(4) function cusparseZcsrsv2_analysis(handle, transA, m, nnz, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.3.62. cusparseScsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseScsrsv2_solve(handle, transA, m, nnz, alpha, descrA, &
           csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.63. cusparseDcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseDcsrsv2_solve(handle, transA, m, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.64. cusparseCcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseCcsrsv2_solve(handle, transA, m, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.65. cusparseZcsrsv2_solve

This function performs the solve phase of csrsv2.

```
integer(4) function cusparseZcsrsv2_solve(handle, transA, m, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, x, y, policy, pBuffer)
  type(cusparseHandle) :: handle
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), x(*), y(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrsv2Info) :: info
  integer :: policy
  character, device :: pBuffer(*)
```

12.3.66. cusparseXcsrsv2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsrsv2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsrsv2Info) :: info
  integer(4), device :: position ! device or host variable
```

12.4. CUSPARSE Level 3 Functions

This section contains interfaces for the level 3 sparse linear algebra functions that perform operations between sparse and dense matrices.

12.4.1. cusparseScsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseScsrmm(handle, transA, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.2. cusparseDcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseDcsrmm(handle, transA, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
```

(continues on next page)

(continued from previous page)

```
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.3. cusparseCcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseCcsrmm(handle, transA, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.4. cusparseZcsrmm

CSRMM performs one of the matrix-matrix operations C := alpha*op(A) * B + beta*C, where op(A) is one of op(A) = A or op(A) = A**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseZcsrmm(handle, transA, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, m, n, k, nnz
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.5. cusparseScsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseScsrmm2(handle, transA, transB, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.6. cusparseDcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseDcsrmm2(handle, transA, transB, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.7. cusparseCcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseCcsrmm2(handle, transA, transB, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.8. cusparseZcsrmm2

CSRMM2 performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(A) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an m x k sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA. B and C are dense matrices.

This function was removed in CUDA 11.0. It should be replaced with a call to cusparseSpMM

```
integer(4) function cusparseZcsrmm2(handle, transA, transB, m, n, k, nnz, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnz
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), B(*), C(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  integer :: ldb, ldc
```

12.4.9. cusparseScsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseScsrsm_analysis(handle, transA, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.4.10. cusparseDcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseDcsrsm_analysis(handle, transA, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
```

(continues on next page)

(continued from previous page)

```
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.4.11. cusparseCcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseCcsrsm_analysis(handle, transA, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.4.12. cusparseZcsrsm_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseZcsrsm_analysis(handle, transA, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: transA, m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.4.13. cusparseScsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseScsrsm_solve(handle, transA, m, n, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(4), device :: X(*), Y(*)
  integer :: ldx, ldy
```

12.4.14. cusparseDcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseDcsrsm_solve(handle, transA, m, n, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  real(8), device :: X(*), Y(*)
  integer :: ldx, ldy
```

12.4.15. cusparseCcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseCcsrsm_solve(handle, transA, m, n, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(4), device :: X(*), Y(*)
  integer :: ldx, ldy
```

12.4.16. cusparseZcsrsm_solve

This function performs the solve phase of csrsm.

```
integer(4) function cusparseZcsrsm_solve(handle, transA, m, n, alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, info, X, ldx, Y, ldy)
  type(cusparseHandle) :: handle
  integer :: transA, m, n
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
  complex(8), device :: X(*), Y(*)
  integer :: ldx, ldy
```

12.4.17. cusparseScsrsm2_bufferSizeExt

This function computes the work buffer size needed for the cusparseScsrsm2 routines.

```
integer(4) function cusparseScsrsm2_bufferSizeExt(handle, algo, transA, transB, m,
↪nrhs, nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(4), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(8) :: pBufferSize
```

12.4.18. cusparseDcsrsm2_bufferSizeExt

This function computes the work buffer size needed for the cusparseDcsrsm2 routines.

```
integer(4) function cusparseDcsrsm2_bufferSizeExt(handle, algo, transA, transB, m,
↪nrhs, nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(8) :: pBufferSize
```

12.4.19. cusparseCcsrsm2_bufferSizeExt

This function computes the work buffer size needed for the cusparseCcsrsm2 routines.

```
integer(4) function cusparseCcsrsm2_bufferSizeExt(handle, algo, transA, transB, m,
↪nrhs, nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(4), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(8) :: pBufferSize
```

12.4.20. cusparseZcsrsm2_bufferSizeExt

This function computes the work buffer size needed for the cusparseZcsrsm2 routines.

```
integer(4) function cusparseZcsrsm2_bufferSizeExt(handle, algo, transA, transB, m,
↪nrhs, nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(8) :: pBufferSize
```

12.4.21. cusparseScsrsm2_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseScsrsm2_analysis(handle, algo, transA, transB, m, nrhs,
↪nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(4), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.22. cusparseDcsrsm2_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseDcsrsm2_analysis(handle, algo, transA, transB, m, nrhs,
↪nnz, alpha, descrA, &
          csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.23. cusparseCcsrsm2_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseCcsrsm2_analysis(handle, algo, transA, transB, m, nrhs,
↪nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(4), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.24. cusparseZcsrsm2_analysis

This function performs the analysis phase of csrsm.

```
integer(4) function cusparseZcsrsm2_analysis(handle, algo, transA, transB, m, nrhs,
↪nnz, alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.25. cusparseScsrsm2_solve

This function performs the solve phase of csrsm2, solving the sparse triangular linear system op(A) * op(X) = alpha * op(B). A is an m x m sparse matrix in CSR storage format; B and X are the right-hand side matrix and the solution matrix, and B is overwritten with X.

```
integer(4) function cusparseScsrsm2_solve(handle, algo, transA, transB, m, nrhs, nnz,
↪alpha, &
          descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(4), device :: B(ldb,*)
```

(continues on next page)

(continued from previous page)

```
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.26. cusparseDcsrsm2_solve

This function performs the solve phase of csrsm2, solving the sparse triangular linear system op(A) * op(X) = alpha * op(B). A is an m x m sparse matrix in CSR storage format; B and X are the right-hand side matrix and the solution matrix, and B is overwritten with X.

```
integer(4) function cusparseDcsrsm2_solve(handle, algo, transA, transB, m, nrhs, nnz,
↪alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.27. cusparseCcsrsm2_solve

This function performs the solve phase of csrsm2, solving the sparse triangular linear system op(A) * op(X) = alpha * op(B). A is an m x m sparse matrix in CSR storage format; B and X are the right-hand side matrix and the solution matrix, and B is overwritten with X.

```
integer(4) function cusparseCcsrsm2_solve(handle, algo, transA, transB, m, nrhs, nnz,
↪alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(4) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(4), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.28. cusparseZcsrsm2_solve

This function performs the solve phase of csrsm2, solving the sparse triangular linear system op(A) * op(X) = alpha * op(B). A is an m x m sparse matrix in CSR storage format; B and X are the right-hand side matrix and the solution matrix, and B is overwritten with X.

```
integer(4) function cusparseZcsrsm2_solve(handle, algo, transA, transB, m, nrhs, nnz,
↪alpha, &
           descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(8) :: alpha ! host or device variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(8), device :: B(ldb,*)
  type(cusparseCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type
```

12.4.29. cusparseXcsrsm2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsrsm2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsrsm2Info) :: info
  integer(4), device :: position ! device or host variable
```

12.4.30. cusparseSbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseSbsrmm(handle, dirA, transA, transB, mb, n, kb, nnzb,
↪alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

12.4.31. cusparseDbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseDbsrmm(handle, dirA, transA, transB, mb, n, kb, nnzb,
↪alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

12.4.32. cusparseCbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseCbsrmm(handle, dirA, transA, transB, mb, n, kb, nnzb,
↪alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

12.4.33. cusparseZbsrmm

BSRMM performs one of the matrix-matrix operations C := alpha*op(A) * op(B) + beta*C, where op(X) is one of op(X) = X or op(X) = X**T, alpha and beta are scalars. A is an mb x kb sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. B and C are dense matrices.

```
integer(4) function cusparseZbsrmm(handle, dirA, transA, transB, mb, n, kb, nnzb,
↪alpha, &
           descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr) :: descrA
```

(continues on next page)

(continued from previous page)

```
  complex(8), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc
```

12.4.34. cusparseSbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseSbsrsm2_bufferSize(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.4.35. cusparseDbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseDbsrsm2_bufferSize(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.4.36. cusparseCbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseCbsrsm2_bufferSize(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.4.37. cusparseZbsrsm2_bufferSize

This function returns the size of the buffer used in bsrsm2.

```
integer(4) function cusparseZbsrsm2_bufferSize(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.4.38. cusparseSbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseSbsrsm2_analysis(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.4.39. cusparseDbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseDbsrsm2_analysis(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.4.40. cusparseCbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseCbsrsm2_analysis(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.4.41. cusparseZbsrsm2_analysis

This function performs the analysis phase of bsrsm2.

```
integer(4) function cusparseZbsrsm2_analysis(handle, dirA, transA, transX, mb, n,
↪nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparseBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.4.42. cusparseSbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseSbsrsm2_solve(handle, dirA, transA, transX, mb, n, nnzb,
↪alpha, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx, y, ldy,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

12.4.43. cusparseDbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseDbsrsm2_solve(handle, dirA, transA, transX, mb, n, nnzb,
↪alpha, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx, y, ldy,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

12.4.44. cusparseCbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseCbsrsm2_solve(handle, dirA, transA, transX, mb, n, nnzb,
↪alpha, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx, y, ldy,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

12.4.45. cusparseZbsrsm2_solve

This function performs the solve phase of bsrsm2.

```
integer(4) function cusparseZbsrsm2_solve(handle, dirA, transA, transX, mb, n, nnzb,
↪alpha, &
          descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx, y, ldy,
↪policy, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
```

(continues on next page)

(continued from previous page)

```
  type(cusparseBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

12.4.46. cusparseXbsrsm2_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrsm2_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrsm2Info) :: info
  integer(4), device :: position ! device or host variable
```

12.4.47. cusparseSgemmi

GEMMI performs the matrix-matrix operations C := alpha*A*B + beta*C where alpha and beta are scalars, A is an m x k dense matrix, B is a k x n sparse matrix, and C is a m x n dense matrix. Fortran programmers should be aware that this function only uses zero-based indexing for B.

This function is deprecated, and will be removed in a future release. It is recommended to use cusparseSpMM instead.

```
integer(4) function cusparseSgemmi(handle, m, n, k, nnz, alpha, &
          A, lda, cscValB, cscColPtrB, cscRowIndB, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: m, n, k, nnz, lda, ldc
  real(4), device :: alpha, beta ! device or host variable
  real(4), device :: A(lda,*)
  real(4), device :: cscValB(*)
  real(4), device :: C(ldc,*)
  integer(4), device :: cscColPtrB(*), cscRowIndB(*)
```

12.4.48. cusparseDgemmi

GEMMI performs the matrix-matrix operations C := alpha*A*B + beta*C where alpha and beta are scalars, A is an m x k dense matrix, B is a k x n sparse matrix, and C is a m x n dense matrix. Fortran programmers should be aware that this function only uses zero-based indexing for B.

This function is deprecated, and will be removed in a future release. It is recommended to use cusparseSpMM instead.

```
integer(4) function cusparseDgemmi(handle, m, n, k, nnz, alpha, &
          A, lda, cscValB, cscColPtrB, cscRowIndB, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: m, n, k, nnz, lda, ldc
  real(8), device :: alpha, beta ! device or host variable
  real(8), device :: A(lda,*)
  real(8), device :: cscValB(*)
```

(continues on next page)

(continued from previous page)

```
  real(8), device :: C(ldc,*)
  integer(4), device :: cscColPtrB(*), cscRowIndB(*)
```

12.4.49. cusparseCgemmi

GEMMI performs the matrix-matrix operations C := alpha*A*B + beta*C where alpha and beta are scalars, A is an m x k dense matrix, B is a k x n sparse matrix, and C is a m x n dense matrix. Fortran programmers should be aware that this function only uses zero-based indexing for B.

This function is deprecated, and will be removed in a future release. It is recommended to use cusparseSpMM instead.

```
integer(4) function cusparseCgemmi(handle, m, n, k, nnz, alpha, &
           A, lda, cscValB, cscColPtrB, cscRowIndB, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: m, n, k, nnz, lda, ldc
  complex(4), device :: alpha, beta ! device or host variable
  complex(4), device :: A(lda,*)
  complex(4), device :: cscValB(*)
  complex(4), device :: C(ldc,*)
  integer(4), device :: cscColPtrB(*), cscRowIndB(*)
```

12.4.50. cusparseZgemmi

GEMMI performs the matrix-matrix operations C := alpha*A*B + beta*C where alpha and beta are scalars, A is an m x k dense matrix, B is a k x n sparse matrix, and C is a m x n dense matrix. Fortran programmers should be aware that this function only uses zero-based indexing for B.

This function is deprecated, and will be removed in a future release. It is recommended to use cusparseSpMM instead.

```
integer(4) function cusparseZgemmi(handle, m, n, k, nnz, alpha, &
           A, lda, cscValB, cscColPtrB, cscRowIndB, beta, C, ldc)
  type(cusparseHandle) :: handle
  integer :: m, n, k, nnz, lda, ldc
  complex(8), device :: alpha, beta ! device or host variable
  complex(8), device :: A(lda,*)
  complex(8), device :: cscValB(*)
  complex(8), device :: C(ldc,*)
  integer(4), device :: cscColPtrB(*), cscRowIndB(*)
```

12.5. CUSPARSE Extra Functions

This section contains interfaces for the extra functions that are used to manipulate sparse matrices.

12.5.1. cusparseXcsrgeamNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

This function was removed in CUDA 11.0. It should be replaced with cusparseXcsrgeam2Nnz

```
integer(4) function cusparseXcsrgeamNnz(handle, m, n, descrA, nnzA, csrRowPtrA,
↪csrColIndA, descrB, nnzB, csrRowPtrB, csrColIndB, descrC, csrRowPtrC,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.5.2. cusparseScsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseScsrgeam2 routines

```
integer(4) function cusparseScsrgeam(handle, m, n, &
          alpha, descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.3. cusparseDcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseDcsrgeam2 routines

```
integer(4) function cusparseDcsrgeam(handle, m, n, &
          alpha, descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.4. cusparseCcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseCcsrgeam2 routines

```
integer(4) function cusparseCcsrgeam(handle, m, n, &
        alpha, descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
        beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.5. cusparseZcsrgeam

CSRGEAM performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeamNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseZcsrgeam2 routines

```
integer(4) function cusparseZcsrgeam(handle, m, n, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.6. cusparseScsrgeam2_bufferSizeExt

This function determines the work buffer size for cusparseScsrgeam2. CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseScsrgeam2_bufferSizeExt(handle, m, n, alpha, descrA, nnzA,
↪ csrValA, csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes
```

12.5.7. cusparseDcsrgeam2_bufferSizeExt

This function determines the work buffer size for cusparseDcsrgeam2. CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseDcsrgeam2_bufferSizeExt(handle, m, n, alpha, &
          descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, beta, descrB, nnzB, &
          csrValB, csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
```

(continues on next page)

(continued from previous page)

```
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes
```

12.5.8. cusparseCcsrgeam2_bufferSizeExt

This function determines the work buffer size for cusparseCcsrgeam2. CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam2_bufferSizeExt(handle, m, n, alpha, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, beta, descrB, nnzB, &
           csrValB, csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes
```

12.5.9. cusparseZcsrgeam2_bufferSizeExt

This function determines the work buffer size for cusparseZcsrgeam2. CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgeam2_bufferSizeExt(handle, m, n, alpha, descrA, &
           nnzA, csrValA, csrRowPtrA, csrColIndA, beta, descrB, nnzB, csrValB, &
           csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes
```

12.5.10. cusparseXcsrgeam2Nnz

cusparseXcsrgeam2Nnz computes the number of nonzero elements which will be produced by CSRGEAM2.

```
integer(4) function cusparseXcsrgeam2Nnz(handle, m, n, descrA, nnzA, csrRowPtrA,
↪csrColIndA, &
          descrB, nnzB, csrRowPtrB, csrColIndB, descrC, csrRowPtrC,
↪nnzTotalDevHostPtr, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrC
  integer(4) :: m, n, nnzA, nnzB
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*)
  integer(c_int) :: nnzTotalDevHostPtr
  character(c_char), device :: pBuffer(*)
```

12.5.11. cusparseScsrgeam2

CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam2(handle, m, n, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, &
          csrColIndA, beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC,
↪csrValC, csrRowPtrC, csrColIndC, pBuffer)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(1), device :: pBuffer ! can be of any type
```

12.5.12. cusparseDcsrgeam2

CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam2(handle, m, n, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, &
          csrColIndA, beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC,
↪csrValC, csrRowPtrC, csrColIndC, pBuffer)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
```

(continues on next page)

(continued from previous page)

```
  real(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(1), device :: pBuffer ! can be of any type
```

12.5.13. cusparseCcsrgeam2

CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam2(handle, m, n, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC, pBuffer)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(4), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(1), device :: pBuffer ! can be of any type
```

12.5.14. cusparseZcsrgeam2

CSRGEAM2 performs the matrix-matrix operation C := alpha * A + beta * B, alpha and beta are scalars. A, B, and C are m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgeam2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgeam2(handle, m, n, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, csrColIndA, &
          beta, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, descrC, csrValC,
↪csrRowPtrC, csrColIndC, pBuffer)
  type(cusparseHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparseMatDescr):: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
  integer(1), device :: pBuffer ! can be of any type
```

12.5.15. cusparseXcsrgemmNnz

cusparseXcsrgemmNnz computes the number of nonzero elements which will be produced by CSRGEMM.

This function was removed in CUDA 11.0. It should be replaced with the cusparseXcsrgemm2Nnz routines

```
integer(4) function cusparseXcsrgemmNnz(handle, transA, transB, m, n, k, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, descrC, csrRowPtrC,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.5.16. cusparseScsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseScsrgemm2 routines

```
integer(4) function cusparseScsrgemm(handle, transA, transB, m, n, k, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, &
           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  real(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.17. cusparseDcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseDcsrgemm2 routines

```
integer(4) function cusparseDcsrgemm(handle, transA, transB, m, n, k, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, &
           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  real(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.18. cusparseCcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseCcsrgemm2 routines

```
integer(4) function cusparseCcsrgemm(handle, transA, transB, m, n, k, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, &
           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.19. cusparseZcsrgemm

CSRGEMM performs the matrix-matrix operation C := op(A) * op(B), where op(X) is one of op(X) = X or op(X) = X**T, A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C}, csrRowPtr{A|B|C}, and csrColInd{A|B|C}. cusparseXcsrgemmNnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

This function was removed in CUDA 11.0. It should be replaced with the cusparseZcsrgemm2 routines

```
integer(4) function cusparseZcsrgemm(handle, transA, transB, m, n, k, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, &
           descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparseMatDescr) :: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrC(*), csrColIndC(*)
```

12.5.20. cusparseScsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseScsrgemm2_bufferSizeExt(handle, m, n, k, alpha, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrRowPtrD, csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  real(4), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

12.5.21. cusparseDcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseDcsrgemm2_bufferSizeExt(handle, m, n, k, alpha, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrRowPtrD, csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  real(8), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

12.5.22. cusparseCcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseCcsrgemm2_bufferSizeExt(handle, m, n, k, alpha, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrRowPtrD, csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  complex(4), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

12.5.23. cusparseZcsrgemm2_bufferSizeExt

This function returns the size of the buffer used in csrgemm2.

```
integer(4) function cusparseZcsrgemm2_bufferSizeExt(handle, m, n, k, alpha, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrRowPtrD, csrColIndD, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  complex(8), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparseMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*)
  type(cusparseCsrgemm2Info) :: info
  integer(8) :: pBufferSizeInBytes
```

12.5.24. cusparseXcsrgemm2Nnz

cusparseXcsrgemm2Nnz computes the number of nonzero elements which will be produced by CSRGEMM2.

```
integer(4) function cusparseXcsrgemm2Nnz(handle, m, n, k, &
           descrA, nnzA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrRowPtrB, csrColIndB, &
           descrD, nnzD, csrRowPtrD, csrColIndD, &
           descrC, csrRowPtrC, nnzTotalDevHostPtr, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer(4) :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*)
  integer(c_int) :: nnzTotalDevHostPtr
  character(c_char), device :: pBuffer(*)
```

12.5.25. cusparseScsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseScsrgemm2(handle, m, n, k, alpha, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrValD, csrRowPtrD, csrColIndD, &
           descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
```

(continues on next page)

(continued from previous page)

```
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*), csrColIndC(*)
  real(4), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  real(4), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.5.26. cusparseDcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseDcsrgemm2(handle, m, n, k, alpha, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrValD, csrRowPtrD, csrColIndD, &
           descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*), csrColIndC(*)
  real(8), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  real(8), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.5.27. cusparseCcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgemm2(handle, m, n, k, alpha, &
           descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, &
           descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, beta, &
           descrD, nnzD, csrValD, csrRowPtrD, csrColIndD, &
           descrC, csrValC, csrRowPtrC, csrColIndC, info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
```

(continues on next page)

(continued from previous page)

```
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*), csrColIndB(*),
↪csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*), csrColIndC(*)
  complex(4), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  complex(4), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.5.28. cusparseZcsrgemm2

CSRGEMM2 performs the matrix-matrix operation C := alpha * A * B + beta * D alpha and beta are scalars. A, B, and C are m x k, k x n, and m x n sparse matrices that are defined in CSR storage format by the three arrays csrVal{A|B|C|D}, csrRowPtr{A|B|C|D}, and csrColInd{A|B|C|D}. cusparseXcsrgemm2Nnz should be used to determine csrRowPtrC and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgemm2(handle, m, n, k, alpha, descrA, nnzA, csrValA,
↪csrRowPtrA, csrColIndA, descrB, nnzB, csrValB, csrRowPtrB, csrColIndB, beta, descrD,
↪ nnzD, csrValD, csrRowPtrD, csrColIndD, descrC, csrValC, csrRowPtrC, csrColIndC,
↪info, pBuffer)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA, descrB, descrD, descrC
  type(cusparseCsrgemm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*),   csrRowPtrB(*), csrColIndB(*),
↪  csrRowPtrD(*), csrColIndD(*), csrRowPtrC(*), csrColIndC(*)
  complex(8), device :: csrValA(*), csrValB(*), csrValD(*), csrValC(*)
  complex(8), device :: alpha, beta ! device or host variable
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.6. CUSPARSE Preconditioning Functions

This section contains interfaces for the preconditioning functions that are used in processing sparse matrices.

12.6.1. cusparseScsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsric0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
```

(continues on next page)

(continued from previous page)

```
  real(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.2. cusparseDcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsric0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.3. cusparseCcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsric0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.4. cusparseZcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an m x n Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsric0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.5. cusparseScsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsrilu0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.6. cusparseDcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsrilu0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.7. cusparseCcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsrilu0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.8. cusparseZcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsrilu0(handle, trans, m, descrA, csrValM, csrRowPtrA,
↪csrColIndA, info)
  type(cusparseHandle) :: handle
  integer(4) :: trans, m
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValM(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

12.6.9. cusparseSgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

This function was removed in CUDA 11.0. It and routines like it should be replaced with the cusparseSgtsv2 variants.

```
integer(4) function cusparseSgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(*), d(*), du(*), B(*)
```

12.6.10. cusparseDgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

This function was removed in CUDA 11.0. It and routines like it should be replaced with the cusparseDgtsv2 variants.

```
integer(4) function cusparseDgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(*), d(*), du(*), B(*)
```

12.6.11. cusparseCgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

This function was removed in CUDA 11.0. It and routines like it should be replaced with the cusparseCgtsv2 variants.

```
integer(4) function cusparseCgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(*), d(*), du(*), B(*)
```

12.6.12. cusparseZgtsv

GTSV computes the solution of a tridiagonal linear system with multiple right hand sides: A * Y = a * x The coefficient matrix A of this tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix X. The solution Y overwrites the righthand-side matrix X on exit.

This function was removed in CUDA 11.0. It and routines like it should be replaced with the cusparseZgtsv2 variants.

```
integer(4) function cusparseZgtsv(handle, m, n, dl, d, du, B, ldb)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(*), d(*), du(*), B(*)
```

12.6.13. cusparseSgtsv2_buffersize

Sgtsv2_buffersize returns the size of the buffer, in bytes, required in Sgtsv2().

```
integer(4) function cusparseSgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.14. cusparseDgtsv2_buffersize

Dgtsv2_buffersize returns the size of the buffer, in bytes, required in Dgtsv2().

```
integer(4) function cusparseDgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.15. cusparseCgtsv2_buffersize

Cgtsv2_buffersize returns the size of the buffer, in bytes, required in Cgtsv2().

```
integer(4) function cusparseCgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.16. cusparseZgtsv2_buffersize

Zgtsv2_buffersize returns the size of the buffer, in bytes, required in Zgtsv2().

```
integer(4) function cusparseZgtsv2_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.17. cusparseSgtsv2

Sgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseSgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.18. cusparseDgtsv2

Dgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseDgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.19. cusparseCgtsv2

Cgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseCgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.20. cusparseZgtsv2

Zgtsv2 computes the solution of a tridiagonal linear system with multiple right hand sides: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseZgtsv2(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.21. cusparseSgtsv2_nopivot_buffersize

Sgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Sgtsv2_nopivot().

```
integer(4) function cusparseSgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪ pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.22. cusparseDgtsv2_nopivot_buffersize

Dgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Dgtsv2_nopivot().

```
integer(4) function cusparseDgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪ pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.23. cusparseCgtsv2_nopivot_buffersize

Cgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Cgtsv2_nopivot().

```
integer(4) function cusparseCgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪ pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.24. cusparseZgtsv2_nopivot_buffersize

Zgtsv2_nopivot_buffersize returns the size of the buffer, in bytes, required in Zgtsv2_nopivot().

```
integer(4) function cusparseZgtsv2_nopivot_bufferSize(handle, m, n, dl, d, du, B, ldb,
↪ pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  integer(8) :: pBufferSizeInBytes
```

12.6.25. cusparseSgtsv2_nopivot

Sgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseSgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.26. cusparseDgtsv2_nopivot

Dgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseDgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  real(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.27. cusparseCgtsv2_nopivot

Cgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseCgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(4), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.28. cusparseZgtsv2_nopivot

Zgtsv2_nopivot computes the solution of a tridiagonal linear system with multiple right hand sides, without pivoting: A * X = B The coefficient matrix A of the tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. The input m is the size of the linear system. The input n is the number or right-hand sides in B.

```
integer(4) function cusparseZgtsv2_nopivot(handle, m, n, dl, d, du, B, ldb, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, ldb
  complex(8), device :: dl(m), d(m), du(m), B(ldb,n)
  character(1), device :: pBuffer(*)
```

12.6.29. cusparseSgtsv2StridedBatch_buffersize

Sgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Sgtsv2StridedBatch().

```
integer(4) function cusparseSgtsv2StridedBatch_bufferSize(handle, m, dl, d, du, x,
↪batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.30. cusparseDgtsv2StridedBatch_buffersize

Dgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Dgtsv2StridedBatch().

```
integer(4) function cusparseDgtsv2StridedBatch_bufferSize(handle, m, dl, d, du, x,
↪batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.31. cusparseCgtsv2StridedBatch_buffersize

Cgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Cgtsv2StridedBatch().

```
integer(4) function cusparseCgtsv2StridedBatch_bufferSize(handle, m, dl, d, du, x,
↪batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.32. cusparseZgtsv2StridedBatch_buffersize

Zgtsv2StridedBatch_buffersize returns the size of the buffer, in bytes, required in Zgtsv2StridedBatch().

```
integer(4) function cusparseZgtsv2StridedBatch_bufferSize(handle, m, dl, d, du, x,
↪batchCount, batchStride, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.33. cusparseSgtsv2StridedBatch

SgtsvStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseSgtsv2StridedBatch(handle, m, dl, d, du, x, batchCount,
↪batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.34. cusparseDgtsv2StridedBatch

DgtsvStridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseDgtsv2StridedBatch(handle, m, dl, d, du, x, batchCount,
↪batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  real(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.35. cusparseCgtsv2StridedBatch

Cgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseCgtsv2StridedBatch(handle, m, dl, d, du, x, batchCount,
↪batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.36. cusparseZgtsv2StridedBatch

Zgtsv2StridedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit.

```
integer(4) function cusparseZgtsv2StridedBatch(handle, m, dl, d, du, x, batchCount,
↪batchStride, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchStride
  complex(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.37. cusparseSgtsvInterleavedBatch_buffersize

SgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in SgtsvInterleaved-Batch().

```
integer(4) function cusparseSgtsvInterleavedBatch_bufferSize(handle, algo, m, dl, d,
↪du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.38. cusparseDgtsvInterleavedBatch_buffersize

DgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in DgtsvInterleavedBatch().

```
integer(4) function cusparseDgtsvInterleavedBatch_bufferSize(handle, algo, m, dl, d,
↪du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.39. cusparseCgtsvInterleavedBatch_buffersize

CgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in CgtsvInterleavedBatch().

```
integer(4) function cusparseCgtsvInterleavedBatch_bufferSize(handle, algo, m, dl, d,
↪du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.40. cusparseZgtsvInterleavedBatch_buffersize

ZgtsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in ZgtsvInterleavedBatch().

```
integer(4) function cusparseZgtsvInterleavedBatch_bufferSize(handle, algo, m, dl, d,
↪du, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.41. cusparseSgtsvInterleavedBatch

SgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the SgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseSgtsvInterleavedBatch(handle, algo, m, dl, d, du, x,
↪batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.42. cusparseDgtsvInterleavedBatch

DgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the DgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseDgtsvInterleavedBatch(handle, algo, m, dl, d, du, x,
↪batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.43. cusparseCgtsvInterleavedBatch

CgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the CgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseCgtsvInterleavedBatch(handle, algo, m, dl, d, du, x,
↪batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.44. cusparseZgtsvInterleavedBatch

ZgtsvInterleavedBatch computes the solution of multiple tridiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each tri-diagonal linear system is defined with three vectors corresponding to its lower (dl), main (d), and upper (du) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from the ZgtsvStridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseZgtsvInterleavedBatch(handle, algo, m, dl, d, du, x,
↪batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.45. cusparseSgpsvInterleavedBatch_buffersize

SgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in SgpsvInterleavedBatch().

```
integer(4) function cusparseSgpsvInterleavedBatch_bufferSize(handle, algo, m, ds, dl,
↪d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.46. cusparseDgpsvInterleavedBatch_buffersize

DgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in DgpsvInterleavedBatch().

```
integer(4) function cusparseDgpsvInterleavedBatch_bufferSize(handle, algo, m, ds, dl,
↪d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.47. cusparseCgpsvInterleavedBatch_buffersize

CgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in CgpsvInterleavedBatch().

```
integer(4) function cusparseCgpsvInterleavedBatch_bufferSize(handle, algo, m, ds, dl,
↪d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.48. cusparseZgpsvInterleavedBatch_buffersize

ZgpsvInterleavedBatch_buffersize returns the size of the buffer, in bytes, required in ZgpsvInterleavedBatch().

```
integer(4) function cusparseZgpsvInterleavedBatch_bufferSize(handle, algo, m, ds, dl,
↪d, du, dw, x, batchCount, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

12.6.49. cusparseSgpsvInterleavedBatch

SgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseSgpsvInterleavedBatch(handle, algo, m, ds, dl, d, du, dw,
↪x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.50. cusparseDgpsvInterleavedBatch

DgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseDgpsvInterleavedBatch(handle, algo, m, ds, dl, d, du, dw,
↪x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  real(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.51. cusparseCgpsvInterleavedBatch

CgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseCgpsvInterleavedBatch(handle, algo, m, ds, dl, d, du, dw,
↪x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(4), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.52. cusparseZgpsvInterleavedBatch

ZgpsvInterleavedBatch computes the solution of multiple pentadiagonal linear systems with multiple right hand sides: A * X = B The coefficient matrix A of each penta-diagonal linear system is defined with five vectors corresponding to its lower (ds, dl), main (d), and upper (du, dw) matrix diagonals; the right-hand sides are stored in the dense matrix B. The solution X overwrites the righthand-side matrix B on exit. This routine differs from StridedBatch routines in that the data for the diagonals, RHS, and solution vectors are interleaved, from one to batchCount, rather than stored one after another. See the cuSPARSE Library document for currently supported algo values.

```
integer(4) function cusparseZgpsvInterleavedBatch(handle, algo, m, ds, dl, d, du, dw,
↪x, batchCount, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: algo, m, batchCount
  complex(8), device :: ds(*), dl(*), d(*), du(*), dw(*), x(*)
  character(1), device :: pBuffer(*)
```

12.6.53. cusparseScsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseScsric02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.54. cusparseDcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseDcsric02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.55. cusparseCcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseCcsric02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.56. cusparseZcsric02_bufferSize

This function returns the size of the buffer used in csric02.

```
integer(4) function cusparseZcsric02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.57. cusparseScsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseScsric02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.58. cusparseDcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseDcsric02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.59. cusparseCcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseCcsric02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.60. cusparseZcsric02_analysis

This function performs the analysis phase of csric02.

```
integer(4) function cusparseZcsric02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.61. cusparseScsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseScsric02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.62. cusparseDcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseDcsric02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.63. cusparseCcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseCcsric02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.64. cusparseZcsric02

CSRIC02 performs the solve phase of computing the incomplete-Cholesky factorization with zero fill-in and no pivoting.

```
integer(4) function cusparseZcsric02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.65. cusparseXcsric02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsric02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsric02Info) :: info
  integer(4), device :: position ! device or host variable
```

12.6.66. cusparseScsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseScsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(4), device :: boost_val ! device or host variable
```

12.6.67. cusparseDcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseDcsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(8), device :: boost_val ! device or host variable
```

12.6.68. cusparseCcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseCcsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(4), device :: boost_val ! device or host variable
```

12.6.69. cusparseZcsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseZcsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(8), device :: boost_val ! device or host variable
```

12.6.70. cusparseScsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseScsrilu02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.71. cusparseDcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseDcsrilu02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.72. cusparseCcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseCcsrilu02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.73. cusparseZcsrilu02_bufferSize

This function returns the size of the buffer used in csrilu02.

```
integer(4) function cusparseZcsrilu02_bufferSize(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.74. cusparseScsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseScsrilu02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.75. cusparseDcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseDcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.76. cusparseCcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseCcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.77. cusparseZcsrilu02_analysis

This function performs the analysis phase of csrilu02.

```
integer(4) function cusparseZcsrilu02_analysis(handle, m, nnz, descrA, csrValA,
↪csrRowPtrA, csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.78. cusparseScsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseScsrilu02(handle, m, nnz, descrA,
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.79. cusparseDcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseDcsrilu02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.80. cusparseCcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseCcsrilu02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.81. cusparseZcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an m x m sparse matrix that is defined in CSR storage format by the three arrays csrValA_valM, csrRowPtrA, and csrColIndA.

```
integer(4) function cusparseZcsrilu02(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseCsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.82. cusparseXcsrilu02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXcsrilu02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseCsrilu02Info) :: info
  integer(4), device :: position ! device or host variable
```

12.6.83. cusparseSbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseSbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.84. cusparseDbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseDbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.85. cusparseCbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseCbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.86. cusparseZbsric02_bufferSize

This function returns the size of the buffer used in bsric02.

```
integer(4) function cusparseZbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.87. cusparseSbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseSbsric02_analysis(handle, dirA, mb, nnzb, descrA, bsrValA,
↪ bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.88. cusparseDbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseDbsric02_analysis(handle, dirA, mb, nnzb, descrA, bsrValA,
↪ bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.89. cusparseCbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseCbsric02_analysis(handle, dirA, mb, nnzb, descrA, bsrValA,
↪ bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.90. cusparseZbsric02_analysis

This function performs the analysis phase of bsric02.

```
integer(4) function cusparseZbsric02_analysis(handle, dirA, mb, nnzb, descrA, bsrValA,
↪ bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.91. cusparseSbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseSbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.92. cusparseDbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseDbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
```

(continues on next page)

(continued from previous page)

```
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.93. cusparseCbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseCbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.94. cusparseZbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseZbsric02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsric02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.95. cusparseXbsric02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsric02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsric02Info) :: info
  integer(4), device :: position ! device or host variable
```

12.6.96. cusparseSbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseSbsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(4), device :: boost_val ! device or host variable
```

12.6.97. cusparseDbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseDbsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  real(8), device :: boost_val ! device or host variable
```

12.6.98. cusparseCbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseCbsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(4), device :: boost_val ! device or host variable
```

12.6.99. cusparseZbsrilu02_numericBoost

This function boosts the value to replace a numerical value in incomplete LU factorization, based on the tol input argument.

```
integer(4) function cusparseZbsrilu02_numericBoost(handle, info, enable_boost, tol,
↪boost_val)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer :: enable_boost
  real(8), device :: tol ! device or host variable
  complex(8), device :: boost_val ! device or host variable
```

12.6.100. cusparseSbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseSbsrilu02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.101. cusparseDbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseDbsrilu02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.102. cusparseCbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseCbsrilu02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.103. cusparseZbsrilu02_bufferSize

This function returns the size of the buffer used in bsrilu02.

```
integer(4) function cusparseZbsrilu02_bufferSize(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.6.104. cusparseSbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseSbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.105. cusparseDbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseDbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.106. cusparseCbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseCbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.107. cusparseZbsrilu02_analysis

This function performs the analysis phase of bsrilu02.

```
integer(4) function cusparseZbsrilu02_analysis(handle, dirA, mb, nnzb, descrA,
↪bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.108. cusparseSbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseSbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.109. cusparseDbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseDbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.110. cusparseCbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseCbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
```

(continues on next page)

(continued from previous page)

```
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.111. cusparseZbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an (mb*blockDim) x (mb*blockDim) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA.

```
integer(4) function cusparseZbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparseBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

12.6.112. cusparseXbsrilu02_zeroPivot

This function returns an error code equal to CUSPARSE_STATUS_ZERO_PIVOT and sets position to j when A(j,j) is either structural zero or numerical zero. Otherwise, position is set to -1.

```
integer(4) function cusparseXbsrilu02_zeroPivot(handle, info, position)
  type(cusparseHandle) :: handle
  type(cusparseBsrilu02Info) :: info
  integer(4), device :: position ! device or host variable
```

12.7. CUSPARSE Reordering Functions

This section contains interfaces for the reordering functions that are used to manipulate sparse matrices.

12.7.1. cusparseScsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseScsrColor(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*), reordering(*)
  real(4), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

12.7.2. cusparseDcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseDcsrColor(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*), reordering(*)
  real(8), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

12.7.3. cusparseCcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseCcsrColor(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*), reordering(*)
  real(4), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

12.7.4. cusparseZcsrColor

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format.

```
integer(4) function cusparseZcsrColor(handle, m, nnz, descrA, csrValA, csrRowPtrA,
↪csrColIndA, fractionToColor, ncolors, coloring, reordering, info)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseColorInfo) :: info
  integer :: m, nnz
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), coloring(*), reordering(*)
  real(8), device :: fractionToColor ! device or host variable
  integer(4), device :: ncolors ! device or host variable
```

12.8. CUSPARSE Format Conversion Functions

This section contains interfaces for the conversion functions that are used to switch between different sparse and dense matrix storage formats.

12.8.1. cusparseSbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseSbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  real(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*), csrColIndC(*)
```

12.8.2. cusparseDbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseDbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  real(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*), csrColIndC(*)
```

12.8.3. cusparseCbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseCbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  complex(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*), csrColIndC(*)
```

12.8.4. cusparseZbsr2csr

This function converts a sparse matrix in BSR format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by the arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseZbsr2csr(handle, dirA, nm, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, mb, nb, blockDim
  type(cusparseMatDescr) :: descrA, descrC
  complex(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*), csrRowPtrC(*), csrColIndC(*)
```

12.8.5. cusparseXcoo2csr

This function converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).

```
integer(4) function cusparseXcoo2csr(handle, cooRowInd, nnz, m, csrRowPtr, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz, m, idxBase
  integer(4), device :: cooRowInd(*), csrRowPtr(*)
```

12.8.6. cusparseScsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseScsc2dense(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.7. cusparseDcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDcsc2dense(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.8. cusparseCcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseCcsc2dense(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.9. cusparseZcsc2dense

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZcsc2dense(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.10. cusparseScsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseScsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

12.8.11. cusparseDcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseDcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  real(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

12.8.12. cusparseCcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseCcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  complex(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

12.8.13. cusparseZcsc2hyb

This function converts the sparse matrix in CSC format that is defined by the three arrays cscValA, cscColPtrA, and cscRowIndA into the sparse matrix A in HYB format.

```
integer(4) function cusparseZcsc2hyb(handle, m, n, descrA, cscValA, cscRowIndA,
↪cscColPtrA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  complex(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
  type(cusparseHybMat) :: hybA
```

12.8.14. cusparseXcsr2bsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXcsr2bsrNnz(handle, dirA, m, n, descrA, csrRowPtrA,
↪csrColIndA, blockDim, descrC, bsrRowPtrC, nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.8.15. cusparseScsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseScsr2bsr(handle, dirA, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  real(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*), bsrColIndC(*)
```

12.8.16. cusparseDcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDcsr2bsr(handle, dirA, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  real(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*), bsrColIndC(*)
```

12.8.17. cusparseCcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCcsr2bsr(handle, dirA, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  complex(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*), bsrColIndC(*)
```

12.8.18. cusparseZcsr2bsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZcsr2bsr(handle, dirA, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, blockDim, descrC, bsrValC, bsrRowPtrC, bsrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dirA, m, n, blockdim
  type(cusparseMatDescr) :: descrA, descrC
  complex(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), bsrRowPtrC(*), bsrColIndC(*)
```

12.8.19. cusparseXcsr2coo

This function converts the array containing the compressed row pointers (corresponding to CSR format) into an array of uncompressed row indices (corresponding to COO format).

```
integer(4) function cusparseXcsr2coo(handle, csrRowPtr, nnz, m, cooRowInd, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: nnz, m, idxBase
  integer(4), device :: csrRowPtr(*), cooRowInd(*)
```

12.8.20. cusparseScsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

This function was removed in CUDA 11.0. Use the cusparseCsr2cscEx2 routines instead.

```
integer(4) function cusparseScsr2csc(handle, m, n, nnz, csrVal, csrRowPtr, csrColInd,
↪cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  real(4), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

12.8.21. cusparseDcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

This function was removed in CUDA 11.0. Use the cusparseCsr2cscEx2 routines instead.

```
integer(4) function cusparseDcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr, csrColInd,
↪cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  real(8), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

12.8.22. cusparseCcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

This function was removed in CUDA 11.0. Use the cusparseCsr2cscEx2 routines instead.

```
integer(4) function cusparseCcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr, csrColInd,
↪cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  complex(4), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

12.8.23. cusparseZcsr2csc

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr.

This function was removed in CUDA 11.0. Use the cusparseCsr2cscEx2 routines instead.

```
integer(4) function cusparseZcsr2csc(handle, m, n, nnz, csrVal, csrRowPtr, csrColInd,
↪cscVal, cscRowInd, cscColPtr, copyValues, idxBase)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, copyValues, idxBase
  complex(8), device :: csrVal(*), cscVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
```

12.8.24. cusparseCsr2cscEx2_bufferSize

This function determines the size of the work buffer needed by cusparseCsr2cscEx2.

```
integer(4) function cusparseScsr2cscEx2_bufferSize(handle, m, n, nnz, csrVal,
↪csrRowPtr, csrColInd, cscVal, &
          cscColPtr, cscRowInd, valType, copyValues, idxBase, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, valType, copyValues, idxBase, alg
  real(4), device :: csrVal(*), cscVal(*)  ! Can be any supported type
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
  integer(8) :: bufferSize
```

12.8.25. cusparseCsr2cscEx2

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrVal, csrRowPtr, and csrColInd into a sparse matrix in CSC format that is defined by arrays cscVal, cscRowInd, and cscColPtr. The type of the arrays is set by the valType argument.

```
integer(4) function cusparseScsr2cscEx2(handle, m, n, nnz, csrVal, csrRowPtr,
↪csrColInd, cscVal, &
          cscColPtr, cscRowInd, valType, copyValues, idxBase, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz, valType, copyValues, idxBase, alg
  real(4), device :: csrVal(*), cscVal(*)  ! Can be any supported type
  integer(4), device :: csrRowPtr(*), csrColInd(*), cscRowInd(*), cscColPtr(*)
  integer(1), device :: buffer ! Can be any type
```

12.8.26. cusparseScsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseScsr2dense(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.27. cusparseDcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDcsr2dense(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.28. cusparseCcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseCcsr2dense(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.29. cusparseZcsr2dense

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZcsr2dense(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), A(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.30. cusparseScsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseScsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.31. cusparseDcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseDcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.32. cusparseCcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseCcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.33. cusparseZcsr2hyb

This function converts the sparse matrix in CSR format that is defined by the three arrays cscValA, cscRowPtrA, and cscColIndA into a sparse matrix in HYB format.

```
integer(4) function cusparseZcsr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA,
↪csrColIndA, hybA, userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.34. cusparseSdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseSdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
↪cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

12.8.35. cusparseDdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseDdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
↪cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

12.8.36. cusparseCdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseCdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
↪cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

12.8.37. cusparseZdense2csc

This function converts the matrix A in dense format into a sparse matrix in CSC format.

```
integer(4) function cusparseZdense2csc(handle, m, n, descrA, A, lda, nnzPerCol,
↪cscValA, cscRowIndA, cscColPtrA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*), cscValA(*)
  integer(4), device :: nnzPerCol(*), cscRowIndA(*), cscColPtrA(*)
```

12.8.38. cusparseSdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseSdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
↪csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

12.8.39. cusparseDdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseDdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
↪csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

12.8.40. cusparseCdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseCdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
↪csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

12.8.41. cusparseZdense2csr

This function converts the matrix A in dense format into a sparse matrix in CSR format.

```
integer(4) function cusparseZdense2csr(handle, m, n, descrA, A, lda, nnzPerRow,
↪csrValA, csrRowPtrA, csrColIndA)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*), csrValA(*)
  integer(4), device :: nnzPerRow(*), csrRowPtrA(*), csrColIndA(*)
```

12.8.42. cusparseSdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseSdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow, hybA,
↪userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

12.8.43. cusparseDdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseDdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow, hybA,
↪userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

12.8.44. cusparseCdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseCdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow, hybA,
↪userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

12.8.45. cusparseZdense2hyb

This function converts the matrix A in dense format into a sparse matrix in HYB format.

```
integer(4) function cusparseZdense2hyb(handle, m, n, descrA, A, lda, nnzPerRow, hybA,
↪userEllWidth, partitionType)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda, userEllWidth, partitionType
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: A(*)
  integer(4), device :: nnzPerRow(*)
```

12.8.46. cusparseShyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseShyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
↪cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
```

(continues on next page)

(continued from previous page)

```
  real(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.47. cusparseDhyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseDhyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
↪cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.48. cusparseChyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseChyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
↪cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.49. cusparseZhyb2csc

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```
integer(4) function cusparseZhyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
↪cscColPtrA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

12.8.50. cusparseShyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseShyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
↪csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.51. cusparseDhyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseDhyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
↪csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.52. cusparseChyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseChyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
↪csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.53. cusparseZhyb2csr

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```
integer(4) function cusparseZhyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
↪csrColIndA)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
```

12.8.54. cusparseShyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseShyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(4), device :: A(*)
  integer(4) :: lda
```

12.8.55. cusparseDhyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseDhyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  real(8), device :: A(*)
  integer(4) :: lda
```

12.8.56. cusparseChyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseChyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(4), device :: A(*)
  integer(4) :: lda
```

12.8.57. cusparseZhyb2dense

This function converts the sparse matrix in HYB format into the matrix A in dense format. The dense matrix A is filled in with the values of the sparse matrix and with zeros elsewhere.

```
integer(4) function cusparseZhyb2dense(handle, descrA, hybA, A, lda)
  type(cusparseHandle) :: handle
  type(cusparseMatDescr) :: descrA
  type(cusparseHybMat) :: hybA
  complex(8), device :: A(*)
  integer(4) :: lda
```

12.8.58. cusparseSnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseSnnz(handle, dirA, m, n, descrA, A, lda, nnzPerRowColumn,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.8.59. cusparseDnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseDnnz(handle, dirA, m, n, descrA, A, lda, nnzPerRowColumn,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.8.60. cusparseCnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseCnnz(handle, dirA, m, n, descrA, A, lda, nnzPerRowColumn,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.8.61. cusparseZnnz

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

```
integer(4) function cusparseZnnz(handle, dirA, m, n, descrA, A, lda, nnzPerRowColumn,
↪nnzTotalDevHostPtr)
  type(cusparseHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
```

12.8.62. cusparseSgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseSgebsr2gebsc_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.63. cusparseDgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseDgebsr2gebsc_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.64. cusparseCgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseCgebsr2gebsc_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.65. cusparseZgebsr2gebsc_bufferSize

This function returns the size of the buffer used in gebsr2gebsc.

```
integer(4) function cusparseZgebsr2gebsc_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, bsrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.66. cusparseSgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseSgebsr2gebsc(handle, mb, nb, nnzb, bsrVal, bsrRowPtr, &
          bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
↪copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  real(4), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

12.8.67. cusparseDgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseDgebsr2gebsc(handle, mb, nb, nnzb, bsrVal, bsrRowPtr, &
          bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
↪copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  real(8), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

12.8.68. cusparseCgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseCgebsr2gebsc(handle, mb, nb, nnzb, bsrVal, bsrRowPtr, &
          bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
↪copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  complex(4), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

12.8.69. cusparseZgebsr2gebsc

This function converts a sparse matrix in general block-CSR storage format to a sparse matrix in general block-CSC storage format.

```
integer(4) function cusparseZgebsr2gebsc(handle, mb, nb, nnzb, bsrVal, bsrRowPtr, &
          bsrColInd, rowBlockDim, colBlockDim, bscVal, bscRowInd, bscColPtr,
↪copyValues, baseIdx, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  complex(8), device :: bscVal(*)
  integer(4), device :: bscRowInd(*), bscColPtr(*)
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)
```

12.8.70. cusparseSgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseSgebsr2gebsr_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, &
           bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC,
↪pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.71. cusparseDgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseDgebsr2gebsr_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, &
           bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC,
↪pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  real(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.72. cusparseCgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseCgebsr2gebsr_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, &
           bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC,
↪pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(4), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.73. cusparseZgebsr2gebsr_bufferSize

This function returns the size of the buffer used in gebsr2gebsrnnz and gebsr2gebsr.

```
integer(4) function cusparseZgebsr2gebsr_bufferSize(handle, mb, nb, nnzb, bsrVal,
↪bsrRowPtr, &
          bsrColInd, rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC,
↪pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: mb, nb, nnzb
  complex(8), device :: bsrVal(*)
  integer(4), device :: bsrRowPtr(*), bsrColInd(*)
  integer(4) :: rowBlockDimA, colBlockDimA, rowBlockDimC, colBlockDimC
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.74. cusparseXgebsr2gebsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXgebsr2gebsrNnz(handle, dir, mb, nb, nnzb, descrA,
↪bsrRowPtrA, &
          bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrRowPtrC, rowBlockDimC,
↪colBlockDimC, nnzTotalDevHostPtr, pBuffer)
  type(cusparseHandle) :: handle
  integer :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*)
  integer :: rowBlockDimC, colBlockDimC
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.8.75. cusparseSgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseSgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, &
          bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC, bsrRowPtrC,
↪bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
```

(continues on next page)

(continued from previous page)

```
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.76. cusparseDgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, &
           bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC, bsrRowPtrC,
↪bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.77. cusparseCgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, &
           bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC, bsrRowPtrC,
↪bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.78. cusparseZgebsr2gebsr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in another general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZgebsr2gebsr(handle, dir, mb, nb, nnzb, descrA, bsrValA,
↪bsrRowPtrA, &
           bsrColIndA, rowBlockDimA, colBlockDimA, descrC, bsrValC, bsrRowPtrC,
↪bsrColIndC, rowBlockDimC, colBlockDimC, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), bsrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.79. cusparseSgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseSgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, &
           rowBlockDimA, colBlockDimA, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

12.8.80. cusparseDgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseDgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, &
           rowBlockDimA, colBlockDimA, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
```

(continues on next page)

(continued from previous page)

```
  type(cusparseMatDescr) :: descrA
  real(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

12.8.81. cusparseCgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseCgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, &
           rowBlockDimA, colBlockDimA, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

12.8.82. cusparseZgebsr2csr

This function converts a sparse matrix in general BSR storage format that is defined by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA into a sparse matrix in CSR format that is defined by arrays csrValC, csrRowPtrC, and csrColIndC.

```
integer(4) function cusparseZgebsr2csr(handle, dir, mb, nb, descrA, bsrValA,
↪bsrRowPtrA, bsrColIndA, &
           rowBlockDimA, colBlockDimA, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: bsrValA(*), csrValC(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: rowBlockDimA, colBlockDimA
  type(cusparseMatDescr) :: descrC
  integer(4), device :: csrRowPtrC(*), csrColIndC(*)
```

12.8.83. cusparseScsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseScsr2gebsr_bufferSize(handle, dir, m, n, descrA, csrVal,
↪csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.84. cusparseDcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseDcsr2gebsr_bufferSize(handle, dir, m, n, descrA, csrVal,
↪csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.85. cusparseCcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseCcsr2gebsr_bufferSize(handle, dir, m, n, descrA, csrVal,
↪csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.86. cusparseZcsr2gebsr_bufferSize

This function returns the size of the buffer used in csr2gebsrnnz and csr2gebsr.

```
integer(4) function cusparseZcsr2gebsr_bufferSize(handle, dir, m, n, descrA, csrVal,
↪csrRowPtr, csrColInd, rowBlockDim, colBlockDim, pBufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: dir, m, n
  complex(8), device :: csrVal(*)
```

(continues on next page)

(continued from previous page)

```
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

12.8.87. cusparseXcsr2gebsrNnz

cusparseXcsrgeamNnz computes the number of nonzero elements which will be produced by CSRGEAM.

```
integer(4) function cusparseXcsr2gebsrNnz(handle, dir, m, n, descrA, csrRowPtrA,
↪csrColIndA, &
           descrC, bsrRowPtrC, rowBlockDimC, colBlockDimC, nnzTotalDevHostPtr,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer :: dir, m, n
  type(cusparseMatDescr) :: descrA
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*)
  integer :: rowBlockDimC, colBlockDimC
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable
  character, device :: pBuffer(*)
```

12.8.88. cusparseScsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseScsr2gebsr(handle, dir, m, n, descrA, csrValA, csrRowPtrA,
↪ csrColIndA, &
           descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.89. cusparseDcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseDcsr2gebsr(handle, dir, m, n, descrA, csrValA, csrRowPtrA,
↪ csrColIndA, &
          descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.90. cusparseCcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseCcsr2gebsr(handle, dir, m, n, descrA, csrValA, csrRowPtrA,
↪ csrColIndA, &
          descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.91. cusparseZcsr2gebsr

This function converts a sparse matrix in CSR storage format that is defined by the three arrays csrValA, csrRowPtrA, and csrColIndA into a sparse matrix in general BSR format that is defined by arrays bsrValC, bsrRowPtrC, and bsrColIndC.

```
integer(4) function cusparseZcsr2gebsr(handle, dir, m, n, descrA, csrValA, csrRowPtrA,
↪ csrColIndA, &
          descrC, bsrValC, bsrRowPtrC, bsrColIndC, rowBlockDimC, colBlockDimC,
↪pBuffer)
  type(cusparseHandle) :: handle
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: dir, mb, nb, nnzb
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrValA(*), bsrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

12.8.92. cusparseCreateIdentityPermutation

This function creates an identity map. The output parameter p represents such map by p = 0:1:(n-1). This function is typically used with coosort, csrsort, cscsort, and csr2csc_indexOnly.

```
integer(4) function cusparseCreateIdentityPermutation(handle, n, p)
  type(cusparseHandle) :: handle
  integer(4) :: n
  integer(4), device :: p(*)
```

12.8.93. cusparseXcoosort_bufferSize

This function returns the size of the buffer used in coosort.

```
integer(4) function cusparseXcoosort_bufferSize(handle, m, n, nnz, cooRows, cooCols,
↪pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*)
  integer(8) :: pBufferSizeInBytes
```

12.8.94. cusparseXcoosortByRow

This function sorts the sparse matrix stored in COO format.

```
integer(4) function cusparseXcoosortByRow(handle, m, n, nnz, cooRows, cooCols, P,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*), P(*)
  character(c_char), device :: pBuffer(*)
```

12.8.95. cusparseXcoosortByColumn

This function sorts the sparse matrix stored in COO format.

```
integer(4) function cusparseXcoosortByColumn(handle, m, n, nnz, cooRows, cooCols, P,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cooRows(*), cooCols(*), P(*)
  character(c_char), device :: pBuffer(*)
```

12.8.96. cusparseXcsrsort_bufferSize

This function returns the size of the buffer used in csrsort.

```
integer(4) function cusparseXcsrsort_bufferSize(handle, m, n, nnz, csrRowInd,
↪csrColInd, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: csrRowInd(*), csrColInd(*)
  integer(8) :: pBufferSizeInBytes
```

12.8.97. cusparseXcsrsort

This function sorts the sparse matrix stored in CSR format.

```
integer(4) function cusparseXcsrsort(handle, m, n, nnz, csrRowInd, csrColInd, P,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: csrRowInd(*), csrColInd(*), P(*)
  character(c_char), device :: pBuffer(*)
```

12.8.98. cusparseXcscsort_bufferSize

This function returns the size of the buffer used in cscsort.

```
integer(4) function cusparseXcscsort_bufferSize(handle, m, n, nnz, cscColPtr,
↪cscRowInd, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cscColPtr(*), cscRowInd(*)
  integer(8) :: pBufferSizeInBytes
```

12.8.99. cusparseXcscsort

This function sorts the sparse matrix stored in CSC format.

```
integer(4) function cusparseXcscsort(handle, m, n, nnz, cscColPtr, cscRowInd, P,
↪pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  integer(4), device :: cscColPtr(*), cscRowInd(*), P(*)
  character(c_char), device :: pBuffer(*)
```

12.8.100. cusparseScsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseScsru2csr_bufferSize(handle, m, n, nnz, csrVal, csrRowPtr,
↪ csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

12.8.101. cusparseDcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseDcsru2csr_bufferSize(handle, m, n, nnz, csrVal, csrRowPtr,
↪ csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

12.8.102. cusparseCcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseCcsru2csr_bufferSize(handle, m, n, nnz, csrVal, csrRowPtr,
↪ csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

12.8.103. cusparseZcsru2csr_bufferSize

This function returns the size of the buffer used in csru2csr.

```
integer(4) function cusparseZcsru2csr_bufferSize(handle, m, n, nnz, csrVal, csrRowPtr,
↪ csrColInd, info, pBufferSizeInBytes)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

12.8.104. cusparseScsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseScsru2csr(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.105. cusparseDcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseDcsru2csr(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.106. cusparseCcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseCcsru2csr(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.107. cusparseZcsru2csr

This function transfers unsorted CSR format to CSR format.

```
integer(4) function cusparseZcsru2csr(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.108. cusparseScsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseScsr2csru(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.109. cusparseDcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseDcsr2csru(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.110. cusparseCcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseCcsr2csru(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.8.111. cusparseZcsr2csru

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```
integer(4) function cusparseZcsr2csru(handle, m, n, nnz, descrA, csrVal, csrRowPtr,
↪csrColInd, info, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparseMatDescr) :: descrA
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparseCsru2csrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

12.9. CUSPARSE Generic API Functions

This section contains interfaces for the generic API functions that perform vector-vector (SpVV), matrix-vector (SpMV), and matrix-matrix (SpMM) operations.

12.9.1. cusparseDenseToSparse_bufferSize

This function returns the size of the workspace needed by cusparseDenseToSparse_analysis(). The value returned is in bytes.

```
integer(4) function cusparseDenseToSparse_bufferSize(handle, matA, matB, alg,
↪bufferSize)
  type(cusparseHandle) :: handle
  type(cusparseDnMatDescr) :: MatA
  type(cusparseSpMatDescr) :: matB
  integer(4) :: alg
  integer(8), intent(out) :: bufferSize
```

12.9.2. cusparseDenseToSparse_analysis

This function updates the number of non-zero elements required in the sparse matrix descriptor matB.

```
integer(4) function cusparseDenseToSparse_analysis(handle, matA, matB, alg, buffer)
  type(cusparseHandle) :: handle
  type(cusparseDnMatDescr) :: matA
  type(cusparseSpMatDescr) :: matB
  integer(4) :: alg
  integer(4), device :: buffer(*)
```

12.9.3. cusparseDenseToSparse_convert

This function fills the sparse matrix values in the area provided in descriptor matB.

```
integer(4) function cusparseDenseToSparse_convert(handle, matA, matB, alg, buffer)
  type(cusparseHandle) :: handle
  type(cusparseDnMatDescr) :: matA
  type(cusparseSpMatDescr) :: matB
  integer(4) :: alg
  integer(4), device :: buffer(*)
```

12.9.4. cusparseSparseToDense_bufferSize

This function returns the size of the workspace needed by cusparseSparseToDense_analysis(). The value returned is in bytes.

```
integer(4) function cusparseSparseToDense_bufferSize(handle, matA, matB, alg,
↪bufferSize)
  type(cusparseHandle) :: handle
  type(cusparseSpMatDescr) :: MatA
  type(cusparseDnMatDescr) :: matB
  integer(4) :: alg
  integer(8), intent(out) :: bufferSize
```

12.9.5. cusparseSparseToDense

This function fills the dense values in the area provided in descriptor matB.

```
integer(4) function cusparseSparseToDense(handle, matA, matB, alg, buffer)
  type(cusparseHandle) :: handle
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB
  integer(4) :: alg
  integer(4), device :: buffer(*)
```

12.9.6. cusparseCreateSpVec

This function initializes the sparse vector descriptor used in the generic API. The type, kind, and rank of the input arguments `indices` and `values` are actually ignored, and taken from the input arguments `idxType` and `valueType`. The vectors are assumed to be contiguous. The `idxBase` argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateSpVec(descr, size, nnz, indices, values, idxType,
↪idxBase, valueType)
  type(cusparseSpVecDescr) :: descr
  integer(8) :: size, nnz
  integer(4), device :: indices(*)
  real(4), device    :: values(*)
  integer(4) :: idxType, idxBase, valueType
```

12.9.7. cusparseDestroySpVec

This function releases the host memory associated with the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseDestroySpVec(descr)
  type(cusparseSpVecDescr) :: descr
```

12.9.8. cusparseSpVecGet

This function returns the fields within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGet(descr, size, nnz, indices, values, idxType,
↪idxBase, valueType)
  type(cusparseSpVecDescr) :: descr
  integer(8) :: size, nnz
  type(c_devptr) :: indices
  type(c_devptr) :: values
  integer(4) :: idxType, idxBase, valueType
```

12.9.9. cusparseSpVecGetIndexBase

This function returns `idxBase` field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGetIndexBase(descr, idxBase)
  type(cusparseSpVecDescr) :: descr
  integer(4) :: idxBase
```

12.9.10. cusparseSpVecGetValues

This function returns the `values` field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseSpVecGetValues(descr, values)
  type(cusparseSpVecDescr) :: descr
  type(c_devptr) :: values
```

12.9.11. cusparseSpVecSetValues

This function sets the `values` field within the sparse vector descriptor used in the generic API. The type, kind and rank of the `values` argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseSpVecSetValues(descr, values)
  type(cusparseSpVecDescr) :: descr
  real(4), device :: values(*)
```

12.9.12. cusparseCreateDnVec

This function initializes the dense vector descriptor used in the generic API. The type, kind, and rank of the `values` input argument is ignored, and taken from the input argument `valueType`. The vector is assumed to be contiguous.

```
integer(4) function cusparseCreateDnVec(descr, size, values, valueType)
  type(cusparseDnVecDescr) :: descr
  integer(8) :: size
  real(4), device    :: values(*)
  integer(4) :: valueType
```

12.9.13. cusparseDestroyDnVec

This function releases the host memory associated with the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDestroyDnVec(descr)
  type(cusparseDnVecDescr) :: descr
```

12.9.14. cusparseDnVecGet

This function returns the fields within the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDnVecGet(descr, size, values, valueType)
  type(cusparseDnVecDescr) :: descr
  integer(8) :: size
  type(c_devptr) :: values
  integer(4) :: valueType
```

12.9.15. cusparseDnVecGetValues

This function returns the `values` field within the dense vector descriptor used in the generic API.

```
integer(4) function cusparseDnVecGetValues(descr, values)
  type(cusparseDnVecDescr) :: descr
  type(c_devptr) :: values
```

12.9.16. cusparseDnVecSetValues

This function sets the `values` field within the dense vector descriptor used in the generic API. The type, kind and rank of the `values` argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseDnVecSetValues(descr, values)
  type(cusparseDnVecDescr) :: descr
  real(4), device :: values(*)
```

12.9.17. cusparseCreateCoo

This function initializes the sparse matrix descriptor in COO format used in the generic API. The type, kind, and rank of the input arguments `cooRowInd`, `cooColInd`, and `cooValues` are actually ignored, and taken from the input arguments `idxType` and `valueType`. The `idxBase` argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCoo(descr, rows, cols, nnz, cooRowInd, cooColInd,
↪cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: cooRowInd(*), cooColInd(*)
  real(4), device    :: cooValues(*)
  integer(4) :: idxType, idxBase, valueType
```

12.9.18. cusparseCreateCooAoS

This function initializes the sparse matrix descriptor in COO format, with Array of Structures layout, used in the generic API. The type, kind, and rank of the input arguments `cooInd` and `cooValues` are actually ignored, and taken from the input arguments `idxType` and `valueType`. The `idxBase` argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCooAoS(descr, rows, cols, nnz, cooInd, cooValues,
↪idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: cooInd(*)
  real(4), device    :: cooValues(*)
  integer(4) :: idxType, idxBase, valueType
```

12.9.19. cusparseCreateCsr

This function initializes the sparse matrix descriptor in CSR format used in the generic API. The type, kind, and rank of the input arguments `csrRowOffsets`, `csrColInd`, and `csrValues` are actually ignored, and taken from the input arguments `csrRowOffsetsType`, `csrColIndType`, and `valueType`. The `idxBase` argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateCsr(descr, rows, cols, nnz, csrRowOffsets,
↪csrColInd, csrValues, &
          csrRowOffsetsType, csrColIndType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  integer(4), device :: csrRowOffsets(*), csrColInd(*)
  real(4), device    :: csrValues(*)
  integer(4) :: csrRowOffsetsType, csrColIndType, idxBase, valueType
```

12.9.20. cusparseCreateBlockedEll

This function initializes the sparse matrix descriptor in Blocked-Ellpack (ELL) format used in the generic API. The type, kind, and rank of the input arguments `ellColInd`, and `ellValues` are actually ignored, and taken from the input arguments `ellIdxType`, and `valueType`. The `idxBase` argument is typically CUSPARSE_INDEX_BASE_ONE in Fortran.

```
integer(4) function cusparseCreateBlockedEll(descr, rows, cols, &
       ellBlockSize, ellCols, ellColInd, ellValues, ellIdxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, ellBlockSize, ellCols
  integer(4), device :: ellColInd(*)
  real(4), device    :: ellValues(*)
  integer(4) :: ellIdxType, idxBase, valueType
```

12.9.21. cusparseDestroySpMat

This function releases the host memory associated with the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseDestroySpMat(descr)
  type(cusparseSpMatDescr) :: descr
```

12.9.22. cusparseCooGet

This function returns the fields from the sparse matrix descriptor in COO format used in the generic API.

```
integer(4) function cusparseCooGet(descr, rows, cols, nnz, cooRowInd, cooColInd,
↪cooValues, idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
```

(continues on next page)

(continued from previous page)

```
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: cooRowInd, cooColInd, cooValues
  integer(4) :: idxType, idxBase, valueType
```

12.9.23. cusparseCooAoSGet

This function returns the fields from the sparse matrix descriptor in COO format, Array of Structures layout, used in the generic API.

```
integer(4) function cusparseCooAoSGet(descr, rows, cols, nnz, cooInd, cooValues,
↪idxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: cooInd, cooValues
  integer(4) :: idxType, idxBase, valueType
```

12.9.24. cusparseCsrGet

This function returns the fields from the sparse matrix descriptor in CSR format used in the generic API.

```
integer(4) function cusparseCsrGet(descr, rows, cols, nnz, csrRowOffsets, csrColInd,
↪csrValues, &
           csrRowOffsetsType, crsColIndType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: csrRowOffsets, csrColInd, csrValues
  integer(4) :: csrRowOffsetsType, csrColIndType, idxBase, valueType
```

12.9.25. cusparseBlockedEllGet

This function returns the fields from the sparse matrix descriptor stored in Blocked-Ellpack (ELL) format.

```
integer(4) function cusparseBlockedEllGet(descr, rows, cols, &
       ellBlockSize, ellCols, ellColInd, ellValues, ellIdxType, idxBase, valueType)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, ellBlockSize, ellCols
  type(c_devptr) :: ellColInd, ellValues
  integer(4) :: ellIdxType, idxBase, valueType
```

12.9.26. cusparseCsrSetPointers

This function sets the pointers in the sparse matrix descriptor in CSR format used in the generic API. Any type for the row offsets, column indices, and values are accepted.

```
integer(4) function cusparseCsrSetPointers(descr, csrRowOffsets, csrColInd, csrValues)
  type(cusparseSpMatDescr) :: descr
  integer(4), device :: csrRowOffsets(*), csrColInd(*)
  real(4), device :: csrValues(*)
```

12.9.27. cusparseCscSetPointers

This function sets the pointers in the sparse matrix descriptor in CSC format used in the generic API. Any type for the column offsets, row indices, and values are accepted.

```
integer(4) function cusparseCscSetPointers(descr, cscColOffsets, cscRowInd, cscValues)
  type(cusparseSpMatDescr) :: descr
  integer(4), device :: cscColOffsets(*), cscRowInd(*)
  real(4), device :: cscValues(*)
```

12.9.28. cusparseSpMatGetFormat

This function returns the `format` field within the sparse matrix descriptor used in the generic API. Valid formats for the generic API are CUSPARSE_FORMAT_CSR, CUSPARSE_FORMAT_COO, and CUSPARSE_FORMAT_COO_AOS.

```
integer(4) function cusparseSpMatGetFormat(descr, format)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: format
```

12.9.29. cusparseSpMatGetIndexBase

This function returns `idxBase` field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetIndexBase(descr, idxBase)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: idxBase
```

12.9.30. cusparseSpMatGetSize

This function returns the sparse matrix size within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetSize(descr, rows, cols, nnz)
  type(cusparseSpMatDescr) :: descr
  integer(8) :: rows, cols, nnz
```

12.9.31. cusparseSpMatGetValues

This function returns the `values` field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetValues(descr, values)
  type(cusparseSpMatDescr) :: descr
  type(c_devptr) :: values
```

12.9.32. cusparseSpMatSetValues

This function sets the `values` field within the sparse matrix descriptor used in the generic API. The type, kind and rank of the `values` argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseSpMatSetValues(descr, values)
  type(cusparseSpMatDescr) :: descr
  real(4), device :: values(*)
```

12.9.33. cusparseSpMatGetStridedBatch

This function returns the `batchCount` field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatGetStridedBatch(descr, batchCount)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
```

12.9.34. cusparseSpMatSetStridedBatch

This function sets the `batchCount` field within the sparse matrix descriptor used in the generic API. It is removed in versions >= CUDA 12.0.

```
integer(4) function cusparseSpMatSetStridedBatch(descr, batchCount)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
```

12.9.35. cusparseCooSetStridedBatch

This function sets the batchCount and batchStride field within the COO sparse matrix descriptor.

```
integer(4) function cusparseCooSetStridedBatch(descr, batchCount, &
      batchStride)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
  integer(8) :: batchStride
```

12.9.36. cusparseCsrSetStridedBatch

This function sets the batchCount and batchStride fields within the CSR sparse matrix descriptor.

```
integer(4) function cusparseCsrSetStridedBatch(descr, batchCount, &
      offsetsBatchStride, columnsValuesBatchStride)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
  integer(8) :: offsetsBatchStride, columnsValuesBatchStride
```

12.9.37. cusparseBsrSetStridedBatch

This function sets the batchCount and batchStride fields within the BSR sparse matrix descriptor.

```
integer(4) function cusparseBsrSetStridedBatch(descr, batchCount, &
      offsetsBatchStride, columnsBatchStride, valuesBatchStride)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
  integer(8) :: offsetsBatchStride, columnsBatchStride
  integer(8) :: valuesBatchStride
```

12.9.38. cusparseSpMatGetAttribute

This function returns the sparse matrix attribute from the sparse matrix descriptor used in the generic API. Attribute can currently be CUSPARSE_SPMAT_FILL_MODE or CUSPARSE_SPMAT_DIAG_TYPE.

```
integer(4) function cusparseSpMatGetAttribute(descr, attribute, data, dataSize)
  type(cusparseSpMatDescr), intent(in) :: descr
  integer(4), intent(in)  :: attribute
  integer(4), intent(out) :: data
  integer(8), intent(in)  :: dataSize
```

12.9.39. cusparseSpMatSetAttribute

This function sets the sparse matrix attribute for the sparse matrix descriptor used in the generic API. Attribute can currently be CUSPARSE_SPMAT_FILL_MODE or CUSPARSE_SPMAT_DIAG_TYPE.

```
integer(4) function cusparseSpMatSetAttribute(descr, attribute, data, dataSize)
  type(cusparseSpMatDescr), intent(out) :: descr
  integer(4), intent(in) :: attribute
  integer(4), intent(in) :: data
  integer(8), intent(in) :: dataSize
```

12.9.40. cusparseCreateDnMat

This function initializes the dense matrix descriptor used in the generic API. The type, kind, and rank of the `values` input argument is ignored, and taken from the input argument `valueType`. The `order` argument in Fortran should normally be CUSPARSE_ORDER_COL.

```
integer(4) function cusparseCreateDnMat(descr, rows, cols, ld, values, valueType,
↪order)
  type(cusparseDnMatDescr) :: descr
  integer(8) :: rows, cols, ld
  real(4), device    :: values(*)
  integer(4), value  :: valueType, order
```

12.9.41. cusparseDestroyDnMat

This function releases the host memory associated with the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDestroyDnMat(descr)
  type(cusparseDnMatDescr) :: descr
```

12.9.42. cusparseDnMatGet

This function returns the fields from the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGet(descr, rows, cols, ld, values, valueType, order)
  type(cusparseDnMatDescr) :: descr
  integer(8) :: rows, cols, ld
  type(c_devptr) :: values
  integer(4) :: valueType, order
```

12.9.43. cusparseDnMatGetValues

This function returns the `values` field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGetValues(descr, values)
  type(cusparseDnMatDescr) :: descr
  type(c_devptr) :: values
```

12.9.44. cusparseDnMatSetValues

This function sets the `values` field within the dense matrix descriptor used in the generic API. The type, kind and rank of the `values` argument is ignored; the type is determined by the valueType field in the descriptor.

```
integer(4) function cusparseDnMatSetValues(descr, values)
  type(cusparseDnMatDescr) :: descr
  real(4), device :: values(*)
```

12.9.45. cusparseDnMatGetStridedBatch

This function returns the `batchCount` field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatGetStridedBatch(descr, batchCount)
  type(cusparseDnMatDescr) :: descr
  integer(4) :: batchCount
```

12.9.46. cusparseDnMatSetStridedBatch

This function sets the `batchCount` field within the dense matrix descriptor used in the generic API.

```
integer(4) function cusparseDnMatSetStridedBatch(descr, batchCount)
  type(cusparseDnMatDescr) :: descr
  integer(4) :: batchCount
```

12.9.47. cusparseSpVV_bufferSize

This function returns the size of the workspace needed by cusparseSpVV(). The value returned is in bytes.

```
integer(4) function cusparseSpVV_bufferSize(handle, opX, vecX, vecY, result,
↪computeType, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opX
  type(cusparseSpVecDescr) :: vecX
  type(cusparseDnVecDescr) :: vecY
```

(continues on next page)

(continued from previous page)

```
  real(4), device    :: result ! device or host variable
  integer(4) :: computeType
  integer(8), intent(out) :: bufferSize
```

12.9.48. cusparseSpVV

This function forms the dot product of a sparse vector `vecX` and a dense vector `vecY`. The `buffer` argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpVV_buffersize(). See the cuSPARSE Library documentation for datatype and `computetype` combinations supported in each release.

```
integer(4) function cusparseSpVV(handle, opX, vecX, vecY, result, computeType, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opX
  type(cusparseSpVecDescr) :: vecX
  type(cusparseDnVecDescr) :: vecY
  real(4), device    :: result ! device or host variable
  integer(4) :: computeType
  integer(4), device :: buffer(*)
```

12.9.49. cusparseSpMV_bufferSize

This function returns the size of the workspace needed by cusparseSpMV(). The value returned is in bytes.

```
integer(4) function cusparseSpMV_bufferSize(handle, opA, alpha, matA, vecX, beta,
↪vecY, computeType, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  integer(8), intent(out) :: bufferSize
```

12.9.50. cusparseSpMV

This function forms the multiplication of a sparse matrix `matA` and a dense vector `vecX` to produce dense vector `vecY`. The `buffer` argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpMV_buffersize(). The type of arguments `alpha` and `beta` should match the `computeType` argument. See the cuSPARSE Library documentation for the datatype, `computeType`, sparse format, and `alg` combinations supported in each release.

```
integer(4) function cusparseSpMV(handle, opA, alpha, matA, vecX, beta, vecY,
↪computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA
```

(continues on next page)

(continued from previous page)

```
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

12.9.51. cusparseSpSV_CreateDescr

This function initializes the sparse matrix descriptor used in solving a sparse triangular system.

```
integer(4) function cusparseSpSV_CreateDescr(descr)
  type(cusparseSpSVDescr) :: descr
```

12.9.52. cusparseSpSV_DestroyDescr

This function frees and destroys the sparse matrix descriptor used in solving a sparse triangular system.

```
integer(4) function cusparseSpSV_DestroyDescr(descr)
  type(cusparseSpSVDescr) :: descr
```

12.9.53. cusparseSpSV_bufferSize

This function returns the size of the workspace needed by cusparseSpSV(). The value returned is in bytes.

```
integer(4) function cusparseSpSV_bufferSize(handle, opA, alpha, matA, vecX, vecY, &
  computeType, alg, spsvDescr, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  type(cusparseSpSVDescr) :: spsvDescr
  integer(8), intent(out) :: bufferSize
```

12.9.54. cusparseSpSV_analysis

This function performs the analysis phase needed by cusparseSpSV().

```
integer(4) function cusparseSpSV_analysis(handle, opA, alpha, matA, vecX, vecY, &
  computeType, alg, spsvDescr, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  type(cusparseSpSVDescr) :: spsvDescr
  integer(4), device :: buffer(*)
```

12.9.55. cusparseSpSV_solve

This function executes the solve phase for the sparse triangular linear system.

```
integer(4) function cusparseSpSV_analysis(handle, opA, alpha, matA, vecX, vecY, &
  computeType, alg, spsvDescr)
  type(cusparseHandle) :: handle
  integer(4) :: opA
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnVecDescr) :: vecX, vecY
  integer(4) :: computeType, alg
  type(cusparseSpSVDescr) :: spsvDescr
```

12.9.56. cusparseSpSV_updateMatrix

This function updates the values in the sparse matrix used by cusparseSpSV(). The value type should match the sparse matrix type. The updates can be to the entire matrix, CUSPARSE_SPSV_UPDATE_GENERAL, or to the diagonal, CUSPARSE_SPSV_UPDATE_DIAGONAL.

```
integer(4) function cusparseSpSV_updateMatrix(handle, spsvDescr, &
      newValues, updatePart)
  type(cusparseHandle) :: handle
  type(cusparseSpSVDescr) :: spsvDescr
  real(4), device :: newValues(*)  ! Any type, same as the Matrix
  integer(4) :: updatePart  ! General or Diagonal
```

12.9.57. cusparseSpMM_bufferSize

This function returns the size of the workspace needed by cusparseSpMM(). The value returned is in bytes.

```
integer(4) function cusparseSpMM_bufferSize(handle, opA, opB, alpha, matA, matB, beta,
↪ matC, computeType, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  integer(8), intent(out) :: bufferSize
```

12.9.58. cusparseSpMM_preprocess

This function can be used to speedup the cusparseSpMM computation. See the cuSPARSE Library documentation for the capabilities supported in each release.

```
integer(4) function cusparseSpMM_preprocess(handle, opA, opB, alpha, matA, matB, beta,
↪ matC, computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

12.9.59. cusparseSpMM

This function forms the multiplication of a sparse matrix `matA` and a dense matrix `matB` to produce dense matrix `matC`. The `buffer` argument can be any type, but the size should be greater than or equal to the size returned from cusparseSpMM_buffersize(). The type of arguments `alpha` and `beta` should match the `computeType` argument. See the cuSPARSE Library documentation for the datatype, `computeType`, sparse format, and `alg` combinations supported in each release.

```
integer(4) function cusparseSpMM(handle, opA, opB, alpha, matA, matB, beta, matC,
↪computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

12.9.60. cusparseSpSM_CreateDescr

This function initializes the sparse matrix descriptor used in solving a sparse triangular system, when there are multiple RHS vectors.

```
integer(4) function cusparseSpSM_CreateDescr(descr)
  type(cusparseSpSMDescr) :: descr
```

12.9.61. cusparseSpSM_DestroyDescr

This function frees and destroys the sparse matrix descriptor used in solving a sparse triangular system.

```
integer(4) function cusparseSpSM_DestroyDescr(descr)
  type(cusparseSpSMDescr) :: descr
```

12.9.62. cusparseSpSM_bufferSize

This function returns the size of the workspace needed by cusparseSpSM(). The value returned is in bytes.

```
integer(4) function cusparseSpSM_bufferSize(handle, opA, opB, alpha, matA, matB, matC,
↪ &
  computeType, alg, spsmDescr, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpSMDescr) :: spsmDescr
  integer(8), intent(out) :: bufferSize
```

12.9.63. cusparseSpSM_analysis

This function performs the analysis phase needed by cusparseSpSM().

```
integer(4) function cusparseSpSM_analysis(handle, opA, opB, alpha, &
  matA, matB, matC, computeType, alg, spsmDescr, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  type(cusparseSpMVDescr) :: spsmDescr
  integer(4), device :: buffer(*)
```

12.9.64. cusparseSpSM_solve

This function executes the solve phase for the sparse triangular linear system.

```
integer(4) function cusparseSpSM_analysis(handle, opA, opB, &
  alpha, matA, matB, matC, computeType, alg, spsmDescr)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha ! device or host variable
  type(cusparseSpMatDescr) :: matA
  type(cusparseDnMatDescr) :: matB, matC
  integer(4) :: computeType, alg
  type(cusparseSpSMDescr) :: spsmDescr
```

12.9.65. cusparseSDDMM_bufferSize

This function returns the size of the workspace needed by cusparseSDDMM(). The value returned is in bytes.

```
integer(4) function cusparseSDDMM_bufferSize(handle, opA, opB, alpha, matA, matB,
↪beta, matC, computeType, alg, bufferSize)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseDnMatDescr) :: matA, matB
  type(cusparseSpMatDescr) :: matC
  integer(4) :: computeType, alg
  integer(8), intent(out) :: bufferSize
```

12.9.66. cusparseSDDMM_preprocess

This function can be used to speedup the cusparseSDDMM computation. See the cuSPARSE Library documentation for the capabilities supported in each release.

```
integer(4) function cusparseSDDMM_preprocess(handle, opA, opB, alpha, matA, matB,
↪beta, matC, computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseDnMatDescr) :: matA, matB
  type(cusparseSpMatDescr) :: matC
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

12.9.67. cusparseSDDMM

This function forms the multiplication of a dense matrix `matA` and a dense matrix `matB`, followed by an element-wise multiplication with the sparsity pattern of matrix `matC`. The `buffer` argument can be any type, but the size should be greater than or equal to the size returned from cusparseSDDMM_buffersize(). The type of arguments `alpha` and `beta` should match the `computeType` argument. See the cuSPARSE Library documentation for the datatype, `computeType`, sparse format, and `alg` combinations supported in each release.

```
integer(4) function cusparseSDDMM(handle, opA, opB, alpha, matA, matB, beta, matC,
↪computeType, alg, buffer)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparseDnMatDescr) :: matA, matB
  type(cusparseSpMatDescr) :: matC
  integer(4) :: computeType, alg
  integer(4), device :: buffer(*)
```

12.9.68. cusparseSpGEMM_CreateDescr

This function initializes the sparse matrix descriptor used in multiplying two sparse matrices together.

```
integer(4) function cusparseSpGEMM_CreateDescr(descr)
  type(cusparseSpGEMMDescr) :: descr
```

12.9.69. cusparseSpGEMM_DestroyDescr

This function frees and destroys the sparse matrix descriptor used in multiplying to sparse matrices together.

```
integer(4) function cusparseSpGEMM_DestroyDescr(descr)
  type(cusparseSpGEMMDescr) :: descr
```

12.9.70. cusparseSpGEMM_workEstimation

This function, along with `cusparseSpGEMM_compute()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_workEstimation()` should pass a null pointer for buffer1. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_workEstimation()` should be made with the actual allocated buffer1 argument.

```
integer(4) function cusparseSpGEMM_workEstimation(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, bufferSize1, buffer1)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
```

(continues on next page)

(continued from previous page)

```
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize1
  integer(4), device :: buffer1
```

12.9.71. cusparseSpGEMM_getNumProducts

This function queries the sparse matrix descriptor for the number of intermediate products.

```
integer(4) function cusparseSpGEMM_getNumProducts(descr, num_prods)
  type(cusparseSpGEMMDescr) :: descr
  integer(8) :: num_prods
```

12.9.72. cusparseSpGEMM_estimateMemory

This function, along with `cusparseSpGEMM_compute()` and `cusparseSpGEMM_workEstimation()`, is used in determining the memory requirements for performing SpGEMM Algorithms 2 and 3.

```
integer(4) function cusparseSpGEMM_estimateMemory(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, &
  chunk_fraction, bufferSize3, buffer3, bufferSize2)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  real(4) :: chunk_fraction
  integer(8) :: bufferSize3
  integer(4), device :: buffer3  ! Any type is okay
  integer(8) :: bufferSize2
```

12.9.73. cusparseSpGEMM_compute

This function, along with `cusparseSpGEMM_workEstimation()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_compute()` should pass a null pointer for buffer2. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_compute()` should be made with the actual allocated buffer2 argument.

```
integer(4) function cusparseSpGEMM_compute(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, bufferSize2, buffer2)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize2
  integer(4), device :: buffer2
```

12.9.74. cusparseSpGEMM_copy

This function, along with `cusparseSpGEMM_workEstimation()` and `cusparseSgGEMM_compute()` are used in performing a sparse matrix multiply. Pointers to the work buffers are kept in the `spgemmDescr` argument.

```
integer(4) function cusparseSpGEMM_copy(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
```

An example of the set of calls needed to perform a sparse matrix multiply follows:

```
! Create a csr descriptor for sparse C.  Sizes unknown
status = cusparseCreateCsr(matC, nd, nd, 0, nullp, nullp, &
           nullp, CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I, &
           CUSPARSE_INDEX_BASE_ONE, CUDA_R_64F)
call printStatus('cusparseCreateCsr', status, CUSPARSE_STATUS_SUCCESS)

status = cusparseSpGEMM_workEstimation(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
         CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
         matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd, bsize, nullp)
call printStatus('cusparseSpGEMM_workEstimation', status, CUSPARSE_STATUS_SUCCESS)

print *,"SpGEMM buffersize1 required: ",bsize
if (bsize.gt.0) allocate(buffer_d(bsize))

status = cusparseSpGEMM_workEstimation(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
         CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
         matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd, bsize, buffer_d)
call printStatus('cusparseSpGEMM_workEstimation', status, CUSPARSE_STATUS_SUCCESS)

status = cusparseSpGEMM_compute(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
         CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
         matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd, bsize2, nullp)
call printStatus('cusparseSpGEMM_compute', status, CUSPARSE_STATUS_SUCCESS)

print *,"SpGEMM buffersize2 required: ",bsize2
if (bsize2.gt.0) allocate(buffer2_d(bsize2))

status = cusparseSpGEMM_compute(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
         CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
         matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd, bsize2, buffer2_d)
call printStatus('cusparseSpGEMM_compute', status, CUSPARSE_STATUS_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
status = cusparseSpMatGetSize(matC, nrows, ncols, nnz)
print *,"SpGEMM C matrix sizes: ",nrows, ncols, nnz
if (nrows.gt.0) allocate(csrRowPtrC_d(nrows+1))
if (nnz.gt.0) allocate(csrColIndC_d(nnz))
if (nnz.gt.0) allocate(csrValDC_d(nnz))

status = cusparseCsrSetPointers(matC, csrRowPtrC_d, csrColIndC_d, csrValDC_d)
call printStatus('cusparseCsrSetPointers', status, CUSPARSE_STATUS_SUCCESS)

status = cusparseSpGEMM_copy(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
         CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
         matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd)
call printStatus('cusparseSpGEMM_copy', status, CUSPARSE_STATUS_SUCCESS)

if (bsize.gt.0) deallocate(buffer_d)
if (bsize2.gt.0) deallocate(buffer2_d)
```

12.9.75. cusparseSpGEMMreuse_workEstimation

This function, along with `cusparseSpGEMM_compute()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_workEstimation()` should pass a null pointer for buffer1. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_workEstimation()` should be made with the actual allocated buffer1 argument.

```
integer(4) function cusparseSpGEMMreuse_workEstimation(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, bufferSize1, buffer1)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize1
  integer(4), device :: buffer1
```

12.9.76. cusparseSpGEMMreuse_nnz

This function, along with `cusparseSpGEMM_compute()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_workEstimation()` should pass a null pointer for buffer1. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_workEstimation()` should be made with the actual allocated buffer1 argument.

```
integer(4) function cusparseSpGEMMreuse_nnz(handle, opA, opB, &
      alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, bufferSize2,
↪buffer2, &
      bufferSize3, buffer3, bufferSize4, buffer4)
  type(cusparseHandle) :: handle
```

(continues on next page)

(continued from previous page)

```
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize2, bufferSize3, bufferSize4
  integer(4), device :: buffer2, buffer3, buffer4
```

12.9.77. cusparseSpGEMMreuse_copy

This function, along with `cusparseSpGEMM_compute()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_workEstimation()` should pass a null pointer for buffer1. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_workEstimation()` should be made with the actual allocated buffer1 argument.

```
integer(4) function cusparseSpGEMMreuse_copy(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr, bufferSize5, buffer5)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize5
  integer(4), device :: buffer5
```

12.9.78. cusparseSpGEMMreuse_compute

This function, along with `cusparseSpGEMM_compute()`, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to `cusparseSgGEMM_workEstimation()` should pass a null pointer for buffer1. The function will return the size requirements needed. Once that space is allocated, another call to `cusparseSpGEMM_workEstimation()` should be made with the actual allocated buffer1 argument.

```
integer(4) function cusparseSpGEMMreuse_compute(handle, opA, opB, &
  alpha, matA, matB, beta, matC, computeType, alg, spgemmDescr)
  type(cusparseHandle) :: handle
  integer(4) :: opA, opB
  real(4) :: alpha, beta ! device or host variable
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
```

Chapter 13. Matrix Solver Runtime Library APIs

This section describes the Fortran interfaces to the CUDA cuSOLVER library. The cuSOLVER functions are only accessible from host code. All of the runtime API routines are integer functions that return an error code; they return a value of CUSOLVER_STATUS_SUCCESS if the call was successful, or another cuSOLVER status return value if there was an error.

Currently we provide Fortran interfaces to the cuSolverDN, the dense LAPACK functions.

Chapter 10 contains examples of accessing the cuSOLVER library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line

```
use cusolverDn
```

to your program unit.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

13.1. CUSOLVER Definitions and Helper Functions

This section contains definitions and data types used in the cuSOLVER library and interfaces to the cuSOLVER helper functions.

The cuSOLVER module contains the following derived type definitions:

```
! Definitions from cusolver_common.h
integer, parameter :: CUSOLVER_VER_MAJOR = 11
integer, parameter :: CUSOLVER_VER_MINOR = 6
integer, parameter :: CUSOLVER_VER_PATCH = 4
```

The cuSOLVER module contains the following enumerations:

```
enum, bind(c)
    enumerator :: CUSOLVER_STATUS_SUCCESS                   = 0
    enumerator :: CUSOLVER_STATUS_NOT_INITIALIZED           = 1
    enumerator :: CUSOLVER_STATUS_ALLOC_FAILED              = 2
    enumerator :: CUSOLVER_STATUS_INVALID_VALUE             = 3
    enumerator :: CUSOLVER_STATUS_ARCH_MISMATCH             = 4
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CUSOLVER_STATUS_MAPPING_ERROR             = 5
    enumerator :: CUSOLVER_STATUS_EXECUTION_FAILED          = 6
    enumerator :: CUSOLVER_STATUS_INTERNAL_ERROR            = 7
    enumerator :: CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED = 8
    enumerator :: CUSOLVER_STATUS_NOT_SUPPORTED             = 9
    enumerator :: CUSOLVER_STATUS_ZERO_PIVOT                = 10
    enumerator :: CUSOLVER_STATUS_INVALID_LICENSE           = 11
    enumerator :: CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED= 12
    enumerator :: CUSOLVER_STATUS_IRS_PARAMS_INVALID        = 13
    enumerator :: CUSOLVER_STATUS_IRS_PARAMS_INVALID_PREC   = 14
    enumerator :: CUSOLVER_STATUS_IRS_PARAMS_INVALID_REFINE = 15
    enumerator :: CUSOLVER_STATUS_IRS_PARAMS_INVALID_MAXITER= 16
    enumerator :: CUSOLVER_STATUS_IRS_INTERNAL_ERROR        = 20
    enumerator :: CUSOLVER_STATUS_IRS_NOT_SUPPORTED         = 21
    enumerator :: CUSOLVER_STATUS_IRS_OUT_OF_RANGE          = 22
    enumerator :: CUSOLVER_STATUS_IRS_NRHS_NOT_SUPPORTED_FOR_REFINE_GMRES=23
    enumerator :: CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED = 25
    enumerator :: CUSOLVER_STATUS_IRS_INFOS_NOT_DESTROYED   = 26
    enumerator :: CUSOLVER_STATUS_IRS_MATRIX_SINGULAR       = 30
    enumerator :: CUSOLVER_STATUS_INVALID_WORKSPACE         = 31
end enum

enum, bind(c)
    enumerator :: CUSOLVER_EIG_TYPE_1 = 1
    enumerator :: CUSOLVER_EIG_TYPE_2 = 2
    enumerator :: CUSOLVER_EIG_TYPE_3 = 3
end enum

enum, bind(c)
    enumerator :: CUSOLVER_EIG_MODE_NOVECTOR = 0
    enumerator :: CUSOLVER_EIG_MODE_VECTOR   = 1
end enum

enum, bind(c)
    enumerator :: CUSOLVER_EIG_RANGE_ALL = 1001
    enumerator :: CUSOLVER_EIG_RANGE_I   = 1002
    enumerator :: CUSOLVER_EIG_RANGE_V   = 1003
end enum

enum, bind(c)
    enumerator :: CUSOLVER_INF_NORM = 104
    enumerator :: CUSOLVER_MAX_NORM = 105
    enumerator :: CUSOLVER_ONE_NORM = 106
    enumerator :: CUSOLVER_FRO_NORM = 107
end enum

enum, bind(c)
    enumerator :: CUSOLVER_IRS_REFINE_NOT_SET          = 1100
    enumerator :: CUSOLVER_IRS_REFINE_NONE             = 1101
    enumerator :: CUSOLVER_IRS_REFINE_CLASSICAL        = 1102
    enumerator :: CUSOLVER_IRS_REFINE_CLASSICAL_GMRES  = 1103
    enumerator :: CUSOLVER_IRS_REFINE_GMRES            = 1104
    enumerator :: CUSOLVER_IRS_REFINE_GMRES_GMRES      = 1105
    enumerator :: CUSOLVER_IRS_REFINE_GMRES_NOPCOND    = 1106
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CUSOLVER_PREC_DD           = 1150
    enumerator :: CUSOLVER_PREC_SS           = 1151
    enumerator :: CUSOLVER_PREC_SHT          = 1152
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVER_R_8I  = 1201
    enumerator :: CUSOLVER_R_8U  = 1202
    enumerator :: CUSOLVER_R_64F = 1203
    enumerator :: CUSOLVER_R_32F = 1204
    enumerator :: CUSOLVER_R_16F = 1205
    enumerator :: CUSOLVER_R_16BF  = 1206
    enumerator :: CUSOLVER_R_TF32  = 1207
    enumerator :: CUSOLVER_R_AP  = 1208
    enumerator :: CUSOLVER_C_8I  = 1211
    enumerator :: CUSOLVER_C_8U  = 1212
    enumerator :: CUSOLVER_C_64F = 1213
    enumerator :: CUSOLVER_C_32F = 1214
    enumerator :: CUSOLVER_C_16F = 1215
    enumerator :: CUSOLVER_C_16BF  = 1216
    enumerator :: CUSOLVER_C_TF32  = 1217
    enumerator :: CUSOLVER_C_AP  = 1218
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVER_ALG_0 = 0
    enumerator :: CUSOLVER_ALG_1 = 1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_STOREV_COLUMNWISE = 0
    enumerator :: CUBLAS_STOREV_ROWWISE    = 1
end enum
```

```
enum, bind(c)
    enumerator :: CUBLAS_DIRECT_FORWARD    = 0
    enumerator :: CUBLAS_DIRECT_BACKWARD   = 1
end enum
```

The cuSOLVERDN module contains the following enumerations and type definitions:

```
type cusolverDnHandle
  type(c_ptr) :: handle
end type

type cusolverDnParams
  type(c_ptr) :: params
end type

type cusolverDnSyevjInfo
  type(c_ptr) :: info
end type

enum, bind(c)
    enumerator :: CUSOLVERDN_GETRF = 0
end enum
```

13.1.1. cusolverDnCreate

This function initializes the cusolverDn library and creates a handle on the cusolverDn context. It must be called before any other cuSolverDn API function is invoked. It allocates hardware resources necessary for accessing the GPU.

```
integer(4) function cusolverDnCreate(handle)
  type(cusolverDnHandle) :: handle
```

13.1.2. cusolverDnDestroy

This function releases CPU-side resources used by the cuSolverDn library.

```
integer(4) function cusolverDnDestroy(handle)
  type(cusolverDnHandle) :: handle
```

13.1.3. cusolverDnCreateParams

This function creates and initializes the 64-bit API structure to default values.

```
integer(4) function cusolverDnCreateParams(params)
  type(cusolverDnParams) :: params
```

13.1.4. cusolverDnDestroyParams

This function releases any resources used by the 64-bit API structure.

```
integer(4) function cusolverDnDestroyParams(params)
  type(cusolverDnParams) :: params
```

13.1.5. cusolverDnCreateSyevjInfo

This function creates and initializes the Syevj API structure to default values.

```
integer(4) function cusolverDnCreateSyevjInfo(info)
  type(cusolverDnSyevjInfo) :: info
```

13.1.6. cusolverDnDestroySyevjInfo

This function releases any resources used by the Syevj API structure.

```
integer(4) function cusolverDnDestroySyevjInfo(info)
  type(cusolverDnSyevjInfo) :: info
```

13.1.7. cusolverDnSetAdvOptions

This function configures the algorithm used in the 64-bit API.

```
integer(4) function cusolverDnSetAdvOptions(params, function, algo)
   type(cusolverDnParams) :: params
   integer(4) :: function, algo
```

13.1.8. cusolverDnGetStream

This function gets the stream used by the cuSolverDn library to execute its routines.

```
integer(4) function cusolverDnGetStream(handle, stream)
  type(cusolverDnHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

13.1.9. cusolverDnSetStream

This function sets the stream to be used by the cuSolverDn library to execute its routines.

```
integer(4) function cusolverDnSetStream(handle, stream)
  type(cusolverDnHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

13.1.10. cusolverDnXsyevjSetTolerance

This function configures the tolerance value used in the Xsyevj functions.

```
integer(4) function cusolverDnXsyevjSetTolerance(info, tolerance)
   type(cusolverSyevjInfo) :: info
   real(8) :: tolerance
```

13.1.11. cusolverDnXsyevjSetMaxSweeps

This function sets the maximum number of sweeps used in the Xsyevj functions.

```
integer(4) function cusolverDnXsyevjSetMaxSweeps(info, sweeps)
   type(cusolverSyevjInfo) :: info
   integer(4) :: sweeps
```

13.1.12. cusolverDnXsyevjSetSortEig

This function sets the sorting behavior used in the Xsyevj functions.

```
integer(4) function cusolverDnXsyevjSetSortEig(info, sort_eig)
   type(cusolverSyevjInfo) :: info
   integer(4) :: sort_eig
```

13.1.13. cusolverDnXsyevjGetResidual

This function provides the residual after the execution of the Xsyevj functions.

```
integer(4) function cusolverDnXsyevjGetResidual(handle, info, residual)
   type(cusolverDnHandle)  :: handle
   type(cusolverSyevjInfo) :: info
   real(8) :: residual
```

13.1.14. cusolverDnXsyevjGetSweeps

This function provides the number of sweeps used after the execution of the Xsyevj functions.

```
integer(4) function cusolverDnXsyevjGetSweeps(handle, info, sweeps)
   type(cusolverDnHandle)  :: handle
   type(cusolverSyevjInfo) :: info
   integer(4) :: sweeps
```

13.2. cusolverDn Legacy API

This section describes the linear solver legacy API of cusolverDn, including Cholesky factorization, LU with partial pivoting, QR factorization, and Bunch-Kaufman (LDLT) factorization.

13.2.1. cusolverDnSpotrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnSpotrf

```
integer function cusolverDnSpotrf_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.2. cusolverDnDpotrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnDpotrf

```
integer function cusolverDnDpotrf_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.3. cusolverDnCpotrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnCpotrf

```
integer function cusolverDnCpotrf_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.4. cusolverDnZpotrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZpotrf

```
integer function cusolverDnZpotrf_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.5. cusolverDnSpotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

```
integer function cusolverDnSpotrf(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.6. cusolverDnDpotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

```
integer function cusolverDnDpotrf(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.7. cusolverDnCpotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

```
integer function cusolverDnCpotrf(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.8. cusolverDnZpotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

```
integer function cusolverDnZpotrf(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.9. cusolverDnSpotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverDnSpotrf`

```
integer function cusolverDnSpotrs(handle, &
    uplo, n, nrhs, A, lda, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.10. cusolverDnDpotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverDnDpotrf`

```
integer function cusolverDnDpotrs(handle, &
    uplo, n, nrhs, A, lda, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.11. cusolverDnCpotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverDnCpotrf`

```
integer function cusolverDnCpotrs(handle, &
    uplo, n, nrhs, A, lda, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.12. cusolverDnZpotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverDnZpotrf`

```
integer function cusolverDnZpotrs(handle, &
    uplo, n, nrhs, A, lda, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.13. cusolverDnSpotrfBatched

This function computes the Cholesky factorization of a sequence of Hermitian positive-definite matrices

```
integer function cusolverDnSpotrfBatched(handle, &
    uplo, n, Aarray, lda, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, batchCount
    type(c_devptr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.14. cusolverDnDpotrfBatched

This function computes the Cholesky factorization of a sequence of Hermitian positive-definite matrices

```
integer function cusolverDnDpotrfBatched(handle, &
    uplo, n, Aarray, lda, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, batchCount
    type(c_devptr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.15. cusolverDnCpotrfBatched

This function computes the Cholesky factorization of a sequence of Hermitian positive-definite matrices

```
integer function cusolverDnCpotrfBatched(handle, &
    uplo, n, Aarray, lda, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, batchCount
    type(c_devptr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.16. cusolverDnZpotrfBatched

This function computes the Cholesky factorization of a sequence of Hermitian positive-definite matrices

```
integer function cusolverDnZpotrfBatched(handle, &
    uplo, n, Aarray, lda, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, batchCount
    type(c_devptr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.17. cusolverDnSpotrsBatched

This function solves a sequence of linear systems resulting from the Cholesky factorization of a sequence of Hermitian positive-definite matrices using `cusolverDnSpotrfBatched`

```
integer function cusolverDnSpotrsBatched(handle, &
    uplo, n, nrhs, Aarray, lda, Barray, ldb, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: n, nrhs, lda, ldb, batchCount
    type(c_devptr), device :: Aarray(*)
    type(c_devptr), device :: Barray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.18. cusolverDnDpotrsBatched

This function solves a sequence of linear systems resulting from the Cholesky factorization of a sequence of Hermitian positive-definite matrices using `cusolverDnDpotrfBatched`

```
integer function cusolverDnDpotrsBatched(handle, &
    uplo, n, nrhs, Aarray, lda, Barray, ldb, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, batchCount
    type(c_devptr), device :: Aarray(*)
    type(c_devptr), device :: Barray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.19. cusolverDnCpotrsBatched

This function solves a sequence of linear systems resulting from the Cholesky factorization of a sequence of Hermitian positive-definite matrices using `cusolverDnCpotrfBatched`

```
integer function cusolverDnCpotrsBatched(handle, &
    uplo, n, nrhs, Aarray, lda, Barray, ldb, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, batchCount
    type(c_devptr), device :: Aarray(*)
    type(c_devptr), device :: Barray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.20. cusolverDnZpotrsBatched

This function solves a sequence of linear systems resulting from the Cholesky factorization of a sequence of Hermitian positive-definite matrices using `cusolverDnZpotrfBatched`

```
integer function cusolverDnZpotrsBatched(handle, &
    uplo, n, nrhs, Aarray, lda, Barray, ldb, devinfo, batchCount)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, batchCount
    type(c_devptr), device :: Aarray(*)
    type(c_devptr), device :: Barray(*)
    integer(4), device, intent(out) :: devinfo
```

13.2.21. cusolverDnSpotri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnSpotri

```
integer function cusolverDnSpotri_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.22. cusolverDnDpotri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnDpotri

```
integer function cusolverDnDpotri_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.23. cusolverDnCpotri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnCpotri

```
integer function cusolverDnCpotri_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.24. cusolverDnZpotri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZpotri

```
integer function cusolverDnZpotri_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.25. cusolverDnSpotri

This function computes the inverse of a positive-definite matrix using the Cholesky factorization computed by `cusolverDnSpotrf`.

```
integer function cusolverDnSpotri(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.26. cusolverDnDpotri

This function computes the inverse of a positive-definite matrix using the Cholesky factorization computed by `cusolverDnDpotrf`.

```
integer function cusolverDnDpotri(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.27. cusolverDnCpotri

This function computes the inverse of a positive-definite matrix using the Cholesky factorization computed by `cusolverDnCpotrf`.

```
integer function cusolverDnCpotri(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.28. cusolverDnZpotri

This function computes the inverse of a positive-definite matrix using the Cholesky factorization computed by `cusolverDnZpotrf`.

```
integer function cusolverDnZpotri(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.29. cusolverDnStrtri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnStrtri`

```
integer function cusolverDnStrtri_buffersize(handle, &
    uplo, diag, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.30. cusolverDnDtrtri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDtrtri`

```
integer function cusolverDnDtrtri_buffersize(handle, &
    uplo, diag, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.31. cusolverDnCtrtri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCtrtri`

```
integer function cusolverDnCtrtri_buffersize(handle, &
    uplo, diag, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.32. cusolverDnZtrtri_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZtrtri`

```
integer function cusolverDnZtrtri_buffersize(handle, &
    uplo, diag, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.33. cusolverDnStrtri

This function computes the inverse of an upper or lower triangular matrix A. It is typically called by `cusolverDnSpotri`.

```
integer function cusolverDnStrtri(handle, &
    uplo, diag, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.34. cusolverDnDtrtri

This function computes the inverse of an upper or lower triangular matrix A. It is typically called by `cusolverDnDpotri`.

```
integer function cusolverDnDtrtri(handle, &
    uplo, diag, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.35. cusolverDnCtrtri

This function computes the inverse of an upper or lower triangular matrix A. It is typically called by `cusolverDnCpotri`.

```
integer function cusolverDnCtrtri(handle, &
    uplo, diag, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.36. cusolverDnZtrtri

This function computes the inverse of an upper or lower triangular matrix A. It is typically called by `cusolverDnZpotri`.

```
integer function cusolverDnZtrtri(handle, &
    uplo, diag, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.37. cusolverDnSlauum_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSlauum`

```
integer function cusolverDnSlauum_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.38. cusolverDnDlauum_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDlauum`

```
integer function cusolverDnDlauum_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.39. cusolverDnClauum_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnClauum`

```
integer function cusolverDnClauum_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.40. cusolverDnZlauum_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZlauum`

```
integer function cusolverDnZlauum_buffersize(handle, &
    uplo, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.41. cusolverDnSlauum

This function computes the product U * U**T or L**T * L. It is typically called by `cusolverDnSpotri`.

```
integer function cusolverDnSlauum(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.42. cusolverDnDlauum

This function computes the product U * U**T or L**T * L. It is typically called by `cusolverDnDpotri`.

```
integer function cusolverDnDlauum(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.43. cusolverDnClauum

This function computes the product U * U**T or L**T * L. It is typically called by `cusolverDnCpotri`.

```
integer function cusolverDnClauum(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.44. cusolverDnZlauum

This function computes the product U * U**T or L**T * L. It is typically called by `cusolverDnZpotri`.

```
integer function cusolverDnZlauum(handle, &
    uplo, n, A, lda, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.45. cusolverDnSgetrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSgetrf`

```
integer function cusolverDnSgetrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.46. cusolverDnDgetrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDgetrf`

```
integer function cusolverDnDgetrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.47. cusolverDnCgetrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCgetrf`

```
integer function cusolverDnCgetrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.48. cusolverDnZgetrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZgetrf

```
integer function cusolverDnZgetrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.49. cusolverDnSgetrf

This function computes the LU factorization of a general mxn matrix

```
integer function cusolverDnSgetrf(handle, &
    m, n, A, lda, workspace, devipiv, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: workspace
    integer(4), device, dimension(*) :: devipiv
    integer(4), device, intent(out) :: devinfo
```

13.2.50. cusolverDnDgetrf

This function computes the LU factorization of a general mxn matrix

```
integer function cusolverDnDgetrf(handle, &
    m, n, A, lda, workspace, devipiv, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: workspace
    integer(4), device, dimension(*) :: devipiv
    integer(4), device, intent(out) :: devinfo
```

13.2.51. cusolverDnCgetrf

This function computes the LU factorization of a general mxn matrix

```
integer function cusolverDnCgetrf(handle, &
    m, n, A, lda, workspace, devipiv, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: workspace
```

(continues on next page)

(continued from previous page)

```
    integer(4), device, dimension(*) :: devipiv
    integer(4), device, intent(out) :: devinfo
```

13.2.52. cusolverDnZgetrf

This function computes the LU factorization of a general mxn matrix

```
integer function cusolverDnZgetrf(handle, &
    m, n, A, lda, workspace, devipiv, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: workspace
    integer(4), device, dimension(*) :: devipiv
    integer(4), device, intent(out) :: devinfo
```

13.2.53. cusolverDnSgetrs

This function solves the system of linear equations resulting from the LU factorization of a matrix using `cusolverDnSgetrf`

```
integer function cusolverDnSgetrs(handle, &
    trans, n, nrhs, A, lda, devipiv, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: trans
    integer(4) :: n, nrhs, lda, ldb
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(4), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.54. cusolverDnDgetrs

This function solves the system of linear equations resulting from the LU factorization of a matrix using `cusolverDnDgetrf`

```
integer function cusolverDnDgetrs(handle, &
    trans, n, nrhs, A, lda, devipiv, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: trans
    integer(4) :: n, nrhs, lda, ldb
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(8), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.55. cusolverDnCgetrs

This function solves the system of linear equations resulting from the LU factorization of a matrix using `cusolverDnCgetrf`

```
integer function cusolverDnCgetrs(handle, &
    trans, n, nrhs, A, lda, devipiv, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: trans
    integer(4) :: n, nrhs, lda, ldb
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(4), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.56. cusolverDnZgetrs

This function solves the system of linear equations resulting from the LU factorization of a matrix using `cusolverDnZgetrf`

```
integer function cusolverDnZgetrs(handle, &
    trans, n, nrhs, A, lda, devipiv, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: trans
    integer(4) :: n, nrhs, lda, ldb
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(8), device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.2.57. cusolverDnSlaswp

This function performs row pivoting. It is typically called by `cusolverDnSgetrf`.

```
integer function cusolverDnSlaswp(handle, &
    n, A, lda, k1, k2, devipiv, incx)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda, k1, k2, incx
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
```

13.2.58. cusolverDnDlaswp

This function performs row pivoting. It is typically called by `cusolverDnDgetrf`.

```
integer function cusolverDnDlaswp(handle, &
    n, A, lda, k1, k2, devipiv, incx)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda, k1, k2, incx
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
```

13.2.59. cusolverDnClaswp

This function performs row pivoting. It is typically called by `cusolverDnCgetrf`.

```
integer function cusolverDnClaswp(handle, &
    n, A, lda, k1, k2, devipiv, incx)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda, k1, k2, incx
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
```

13.2.60. cusolverDnZlaswp

This function performs row pivoting. It is typically called by `cusolverDnZgetrf`.

```
integer function cusolverDnZlaswp(handle, &
    n, A, lda, k1, k2, devipiv, incx)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda, k1, k2, incx
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
```

13.2.61. cusolverDnSgeqrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSgeqrf`

```
integer function cusolverDnSgeqrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.62. cusolverDnDgeqrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDgeqrf`

```
integer function cusolverDnDgeqrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.63. cusolverDnCgeqrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCgeqrf`

```
integer function cusolverDnCgeqrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.64. cusolverDnZgeqrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZgeqrf`

```
integer function cusolverDnZgeqrf_buffersize(handle, &
    m, n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.65. cusolverDnSgeqrf

This function computes the QR factorization of an mxn matrix

```
integer function cusolverDnSgeqrf(handle, &
    m, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.66. cusolverDnDgeqrf

This function computes the QR factorization of an mxn matrix

```
integer function cusolverDnDgeqrf(handle, &
    m, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.67. cusolverDnCgeqrf

This function computes the QR factorization of an mxn matrix

```
integer function cusolverDnCgeqrf(handle, &
    m, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.68. cusolverDnZgeqrf

This function computes the QR factorization of an mxn matrix

```
integer function cusolverDnZgeqrf(handle, &
    m, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.69. cusolverDnSorgqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSorgqr`

```
integer function cusolverDnSorgqr_buffersize(handle, &
    m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda
```

(continues on next page)

(continued from previous page)

```
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.2.70. cusolverDnDorgqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDorgqr`

```
integer function cusolverDnDorgqr_buffersize(handle, &
    m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.2.71. cusolverDnCorgqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCorgqr`

```
integer function cusolverDnCorgqr_buffersize(handle, &
    m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.2.72. cusolverDnZorgqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZorgqr`

```
integer function cusolverDnZorgqr_buffersize(handle, &
    m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.2.73. cusolverDnSorgqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix

```
integer function cusolverDnSorgqr(handle, &
    m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.74. cusolverDnDorgqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix

```
integer function cusolverDnDorgqr(handle, &
    m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.75. cusolverDnCorgqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix

```
integer function cusolverDnCorgqr(handle, &
    m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.76. cusolverDnZorgqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix

```
integer function cusolverDnZorgqr(handle, &
    m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, k, lda, lwork
    complex(8), device, dimension(lda,*) :: A
```

(continues on next page)

(continued from previous page)

```
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.77. cusolverDnSormqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSormqr`

```
integer function cusolverDnSormqr_buffersize(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.2.78. cusolverDnDormqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDormqr`

```
integer function cusolverDnDormqr_buffersize(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.2.79. cusolverDnCormqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCormqr`

```
integer function cusolverDnCormqr_buffersize(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.2.80. cusolverDnZormqr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZormqr

```
integer function cusolverDnZormqr_buffersize(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.2.81. cusolverDnSormqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnSormqr(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(ldc,*) :: C
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.82. cusolverDnDormqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnDormqr(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(ldc,*) :: C
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.83. cusolverDnCormqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnCormqr(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(ldc,*) :: C
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.84. cusolverDnZormqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnZormqr(handle, &
    side, trans, m, n, k, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, trans
    integer(4) :: m, n, k, lda, ldc, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(ldc,*) :: C
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.85. cusolverDnSsytrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsytrf`

```
integer function cusolverDnSsytrf_buffersize(handle, &
    n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.86. cusolverDnDsytrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsytrf`

```
integer function cusolverDnDsytrf_buffersize(handle, &
    n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.87. cusolverDnCsytrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCsytrf`

```
integer function cusolverDnCsytrf_buffersize(handle, &
    n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.88. cusolverDnZsytrf_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZsytrf`

```
integer function cusolverDnZsytrf_buffersize(handle, &
    n, A, lda, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

13.2.89. cusolverDnSsytrf

This function computes the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix

```
integer function cusolverDnSsytrf(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
```

(continues on next page)

(continued from previous page)

```
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.90. cusolverDnDsytrf

This function computes the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix

```
integer function cusolverDnDsytrf(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.91. cusolverDnCsytrf

This function computes the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix

```
integer function cusolverDnCsytrf(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.92. cusolverDnZsytrf

This function computes the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix

```
integer function cusolverDnZsytrf(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.93. cusolverDnSsytrs_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsytrs`

```
integer function cusolverDnSsytrs_bufferSize(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(4), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: lwork
```

13.2.94. cusolverDnDsytrs_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsytrs`

```
integer function cusolverDnDsytrs_bufferSize(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(8), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: lwork
```

13.2.95. cusolverDnCsytrs_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCsytrs`

```
integer function cusolverDnCsytrs_bufferSize(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(4), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: lwork
```

13.2.96. cusolverDnZsytrs_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZsytrs`

```
integer function cusolverDnZsytrs_bufferSize(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(8), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: lwork
```

13.2.97. cusolverDnSsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnSsytrf`

```
integer function cusolverDnSsytrs(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, lwork
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.98. cusolverDnDsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnDsytrf`

```
integer function cusolverDnDsytrs(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, lwork
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.99. cusolverDnCsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnCsytrf`

```
integer function cusolverDnCsytrs(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, lwork
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(4), device, dimension(ldb,*) :: B
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.100. cusolverDnZsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnZsytrf`

```
integer function cusolverDnZsytrs(handle, &
    uplo, n, nrhs, A, lda, devipiv, B, ldb, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, nrhs, lda, ldb, lwork
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(8), device, dimension(ldb,*) :: B
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.101. cusolverDnSsytri_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsytri`

```
integer function cusolverDnSsytri_bufferSize(handle, &
    uplo, n, A, lda, devipiv, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    integer(4), intent(out) :: lwork
```

13.2.102. cusolverDnDsytri_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsytri`

```
integer function cusolverDnDsytri_bufferSize(handle, &
    uplo, n, A, lda, devipiv, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    integer(4), intent(out) :: lwork
```

13.2.103. cusolverDnCsytri_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCsytri`

```
integer function cusolverDnCsytri_bufferSize(handle, &
    uplo, n, A, lda, devipiv, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    integer(4), intent(out) :: lwork
```

13.2.104. cusolverDnZsytri_bufferSize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZsytri`

```
integer function cusolverDnZsytri_bufferSize(handle, &
    uplo, n, A, lda, devipiv, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    integer(4), intent(out) :: lwork
```

13.2.105. cusolverDnSsytri

This function inverts the matrix resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnSsytrf`

```
integer function cusolverDnSsytri(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.106. cusolverDnDsytri

This function inverts the matrix resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnDsytrf`

```
integer function cusolverDnDsytri(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.107. cusolverDnCsytri

This function inverts the matrix resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnCsytrf`

```
integer function cusolverDnCsytri(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.2.108. cusolverDnZsytri

This function inverts the matrix resulting from the Bunch-Kaufman factorization of an nxn symmetric indefinite matrix using `cusolverDnZsytrf`

```
integer function cusolverDnZsytri(handle, &
    uplo, n, A, lda, devipiv, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    integer(4), device, dimension(*) :: devipiv
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3. cusolverDn Legacy Eigenvalue Solver API

This section describes the eigenvalue solver legacy API of cusolverDn, including bidiagonalization and SVD.

13.3.1. cusolverDnSgebrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSgebrd`

```
integer function cusolverDnSgebrd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.2. cusolverDnDgebrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDgebrd`

```
integer function cusolverDnDgebrd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.3. cusolverDnCgebrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCgebrd`

```
integer function cusolverDnCgebrd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.4. cusolverDnZgebrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZgebrd`

```
integer function cusolverDnZgebrd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.5. cusolverDnSgebrd

This function reduces a general mxn matrix A to an upper or lower bidiagonal form B by an orthogonal transformation QH * A * P = B

```
integer function cusolverDnSgebrd(handle, &
    m, n, A, lda, D, E, TauQ, TauP, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: D
    real(4), device, dimension(*) :: E
    real(4), device, dimension(*) :: TauQ
    real(4), device, dimension(*) :: TauP
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.6. cusolverDnDgebrd

This function reduces a general mxn matrix A to an upper or lower bidiagonal form B by an orthogonal transformation QH * A * P = B

```
integer function cusolverDnDgebrd(handle, &
    m, n, A, lda, D, E, TauQ, TauP, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
```

(continues on next page)

(continued from previous page)

```
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: D
    real(8), device, dimension(*) :: E
    real(8), device, dimension(*) :: TauQ
    real(8), device, dimension(*) :: TauP
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.7. cusolverDnCgebrd

This function reduces a general mxn matrix A to an upper or lower bidiagonal form B by an orthogonal transformation QH * A * P = B

```
integer function cusolverDnCgebrd(handle, &
    m, n, A, lda, D, E, TauQ, TauP, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: D
    complex(4), device, dimension(*) :: E
    complex(4), device, dimension(*) :: TauQ
    complex(4), device, dimension(*) :: TauP
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.8. cusolverDnZgebrd

This function reduces a general mxn matrix A to an upper or lower bidiagonal form B by an orthogonal transformation QH * A * P = B

```
integer function cusolverDnZgebrd(handle, &
    m, n, A, lda, D, E, TauQ, TauP, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: D
    complex(8), device, dimension(*) :: E
    complex(8), device, dimension(*) :: TauQ
    complex(8), device, dimension(*) :: TauP
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.9. cusolverDnSorgbr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSorgbr`

```
integer function cusolverDnSorgbr_buffersize(handle, &
    side, m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.10. cusolverDnDorgbr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDorgbr`

```
integer function cusolverDnDorgbr_buffersize(handle, &
    side, m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.11. cusolverDnCungbr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCungbr`

```
integer function cusolverDnCungbr_buffersize(handle, &
    side, m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.12. cusolverDnZungbr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZungbr`

```
integer function cusolverDnZungbr_buffersize(handle, &
    side, m, n, k, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.13. cusolverDnSorgbr

This function generates one of the unitary matrices Q or P**H determined by `cusolverDnSgebrd`.

```
integer function cusolverDnSorgbr(handle, &
    side, m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.14. cusolverDnDorgbr

This function generates one of the unitary matrices Q or P**H determined by `cusolverDnDgebrd`.

```
integer function cusolverDnDorgbr(handle, &
    side, m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.15. cusolverDnCungbr

This function generates one of the unitary matrices Q or P**H determined by `cusolverDnCgebrd`.

```
integer function cusolverDnCungbr(handle, &
    side, m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.16. cusolverDnZungbr

This function generates one of the unitary matrices Q or P**H determined by `cusolverDnZgebrd`.

```
integer function cusolverDnZungbr(handle, &
    side, m, n, k, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: m, n, k, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.17. cusolverDnSsytrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsytrd`

```
integer function cusolverDnSsytrd_buffersize(handle, &
    uplo, n, A, lda, D, E, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: D
    real(4), device, dimension(*) :: E
    real(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.18. cusolverDnDsytrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsytrd`

```
integer function cusolverDnDsytrd_buffersize(handle, &
    uplo, n, A, lda, D, E, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: D
    real(8), device, dimension(*) :: E
    real(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.19. cusolverDnChetrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnChetrd`

```
integer function cusolverDnChetrd_buffersize(handle, &
    uplo, n, A, lda, D, E, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: D
    complex(4), device, dimension(*) :: E
    complex(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.20. cusolverDnZhetrd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZhetrd`

```
integer function cusolverDnZhetrd_buffersize(handle, &
    uplo, n, A, lda, D, E, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: D
    complex(8), device, dimension(*) :: E
    complex(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.21. cusolverDnSsytrd

This function reduces a general symmetric or Hermitian nxn matrix to real symmetric tridiagonal form.

```
integer function cusolverDnSsytrd(handle, &
    uplo, n, A, lda, D, E, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: D
    real(4), device, dimension(*) :: E
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.22. cusolverDnDsytrd

This function reduces a general symmetric or Hermitian nxn matrix to real symmetric tridiagonal form.

```
integer function cusolverDnDsytrd(handle, &
    uplo, n, A, lda, D, E, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: D
    real(8), device, dimension(*) :: E
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.23. cusolverDnChetrd

This function reduces a general symmetric or Hermitian nxn matrix to real symmetric tridiagonal form.

```
integer function cusolverDnChetrd(handle, &
    uplo, n, A, lda, D, E, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: D
    complex(4), device, dimension(*) :: E
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.24. cusolverDnZhetrd

This function reduces a general symmetric or Hermitian nxn matrix to real symmetric tridiagonal form.

```
integer function cusolverDnZhetrd(handle, &
    uplo, n, A, lda, D, E, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: D
    complex(8), device, dimension(*) :: E
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.25. cusolverDnSormtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSormtr`

```
integer function cusolverDnSormtr_buffersize(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.3.26. cusolverDnDormtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDormtr`

```
integer function cusolverDnDormtr_buffersize(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.3.27. cusolverDnCunmtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCunmtr`

```
integer function cusolverDnCunmtr_buffersize(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.3.28. cusolverDnZunmtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZunmtr`

```
integer function cusolverDnZunmtr_buffersize(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(ldc,*) :: C
    integer(4) :: lwork
```

13.3.29. cusolverDnSormtr

This function generates the unitary matrix Q formed by a sequence of elementary reflection vectors and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnSormtr(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(ldc,*) :: C
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.30. cusolverDnDormtr

This function generates the unitary matrix Q formed by a sequence of elementary reflection vectors and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnDormtr(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(ldc,*) :: C
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.31. cusolverDnCunmtr

This function generates the unitary matrix Q formed by a sequence of elementary reflection vectors and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnCunmtr(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(ldc,*) :: C
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.32. cusolverDnZunmtr

This function generates the unitary matrix Q formed by a sequence of elementary reflection vectors and overwrites the array C, based on the side and trans arguments.

```
integer function cusolverDnZunmtr(handle, &
    side, uplo, trans, m, n, A, lda, tau, C, ldc, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(4) :: m, n, lda, ldc, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(ldc,*) :: C
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.33. cusolverDnSorgtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSorgtr`

```
integer function cusolverDnSorgtr_buffersize(handle, &
    uplo, n, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.34. cusolverDnDorgtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDorgtr`

```
integer function cusolverDnDorgtr_buffersize(handle, &
    uplo, n, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.35. cusolverDnCungtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCungtr`

```
integer function cusolverDnCungtr_buffersize(handle, &
    uplo, n, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.36. cusolverDnZungtr_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZungtr`

```
integer function cusolverDnZungtr_buffersize(handle, &
    uplo, n, A, lda, tau, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    integer(4) :: lwork
```

13.3.37. cusolverDnSorgtr

This function generates a unitary matrix Q defined as the product of n-1 elementary reflectors of order n, produced by `cusolverDnSsytrd`.

```
integer function cusolverDnSorgtr(handle, &
    uplo, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.38. cusolverDnDorgtr

This function generates a unitary matrix Q defined as the product of n-1 elementary reflectors of order n, produced by `cusolverDnDsytrd`.

```
integer function cusolverDnDorgtr(handle, &
    uplo, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: tau
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.39. cusolverDnCungtr

This function generates a unitary matrix Q defined as the product of n-1 elementary reflectors of order n, produced by `cusolverDnCsytrd`.

```
integer function cusolverDnCungtr(handle, &
    uplo, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: tau
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.40. cusolverDnZungtr

This function generates a unitary matrix Q defined as the product of n-1 elementary reflectors of order n, produced by `cusolverDnZsytrd`.

```
integer function cusolverDnZungtr(handle, &
    uplo, n, A, lda, tau, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.41. cusolverDnSgesvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSgesvd`

```
integer function cusolverDnSgesvd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.42. cusolverDnDgesvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDgesvd`

```
integer function cusolverDnDgesvd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.43. cusolverDnCgesvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCgesvd`

```
integer function cusolverDnCgesvd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.44. cusolverDnZgesvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZgesvd`

```
integer function cusolverDnZgesvd_buffersize(handle, &
    m, n, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n
    integer(4) :: lwork
```

13.3.45. cusolverDnSgesvd

This function computes the singular value decomposition (SVD) of an mxn matrix A and the corresponding left and/or right singular vectors. CusolverDnSgesvd only supports m >= n.

```
integer function cusolverDnSgesvd(handle, &
    jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt, workspace, lwork, rwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, ldu, ldvt, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: S
    real(4), device, dimension(ldu,*) :: U
    real(4), device, dimension(ldvt,*) :: VT
    real(4), device, dimension(lwork) :: workspace
```

(continues on next page)

(continued from previous page)

```
    real(4), device, dimension(lwork) :: rwork
    integer(4), device, intent(out) :: devinfo
```

13.3.46. cusolverDnDgesvd

This function computes the singular value decomposition (SVD) of an mxn matrix A and the corresponding left and/or right singular vectors. CusolverDnDgesvd only supports m >= n.

```
integer function cusolverDnDgesvd(handle, &
    jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt, workspace, lwork, rwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, ldu, ldvt, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: S
    real(8), device, dimension(ldu,*) :: U
    real(4), device, dimension(ldvt,*) :: VT
    real(8), device, dimension(lwork) :: workspace
    real(8), device, dimension(lwork) :: rwork
    integer(4), device, intent(out) :: devinfo
```

13.3.47. cusolverDnCgesvd

This function computes the singular value decomposition (SVD) of an mxn matrix A and the corresponding left and/or right singular vectors. CusolverDnCgesvd only supports m >= n.

```
integer function cusolverDnCgesvd(handle, &
    jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt, workspace, lwork, rwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, ldu, ldvt, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: S
    complex(4), device, dimension(ldu,*) :: U
    complex(4), device, dimension(ldvt,*) :: VT
    complex(4), device, dimension(lwork) :: workspace
    real(4), device, dimension(lwork) :: rwork
    integer(4), device, intent(out) :: devinfo
```

13.3.48. cusolverDnZgesvd

This function computes the singular value decomposition (SVD) of an mxn matrix A and the corresponding left and/or right singular vectors. CusolverDnZgesvd only supports m >= n.

```
integer function cusolverDnZgesvd(handle, &
    jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt, workspace, lwork, rwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: m, n, lda, ldu, ldvt, lwork
    complex(8), device, dimension(lda,*) :: A
```

(continues on next page)

(continued from previous page)

```
    complex(8), device, dimension(*) :: S
    complex(8), device, dimension(ldu,*) :: U
    complex(8), device, dimension(ldvt,*) :: VT
    complex(8), device, dimension(lwork) :: workspace
    real(8), device, dimension(lwork) :: rwork
    integer(4), device, intent(out) :: devinfo
```

13.3.49. cusolverDnSsyevd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnSsyevd

```
integer function cusolverDnSsyevd_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: W
    integer(4) :: lwork
```

13.3.50. cusolverDnDsyevd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnDsyevd

```
integer function cusolverDnDsyevd_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: W
    integer(4) :: lwork
```

13.3.51. cusolverDnCheevd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnCheevd

```
integer function cusolverDnCheevd_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(*) :: W
    integer(4) :: lwork
```

13.3.52. cusolverDnZheevd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZheevd

```
integer function cusolverDnZheevd_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(*) :: W
    integer(4) :: lwork
```

13.3.53. cusolverDnSsyevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnSsyevd(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.54. cusolverDnDsyevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnDsyevd(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.55. cusolverDnCheevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnCheevd(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.56. cusolverDnZheevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnZheevd(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.57. cusolverDnSsyevdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnSsyevdx

```
integer function cusolverDnSsyevdx_buffersize(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu
    real(4) :: vl, vu
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.58. cusolverDnDsyevdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsyevdx`

```
integer function cusolverDnDsyevdx_buffersize(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu
    real(8) :: vl, vu
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.59. cusolverDnCheevdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnCheevdx`

```
integer function cusolverDnCheevdx_buffersize(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu
    real(4) :: vl, vu
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.60. cusolverDnZheevdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZheevdx`

```
integer function cusolverDnZheevdx_buffersize(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu
    real(8) :: vl, vu
    complex(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.61. cusolverDnSsyevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnSsyevdx(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu, lwork
    real(4) :: vl, vu
    real(4), device, dimension(lda,*) :: A
    integer(4), intent(out) :: meig
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.62. cusolverDnDsyevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnDsyevdx(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu, lwork
    real(8) :: vl, vu
    real(8), device, dimension(lda,*) :: A
    integer(4), intent(out) :: meig
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.63. cusolverDnCheevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnCheevdx(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu, lwork
    real(4) :: vl, vu
    complex(4), device, dimension(lda,*) :: A
    integer(4), intent(out) :: meig
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.64. cusolverDnZheevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnZheevdx(handle, &
    jobz, range, uplo, n, A, lda, vl, vu, il, iu, meig, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, range, uplo
    integer(4) :: n, lda, il, iu, lwork
    real(8) :: vl, vu
    complex(8), device, dimension(lda,*) :: A
    integer(4), intent(out) :: meig
    real(8), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.65. cusolverDnSsyevj_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnSsyevj

```
integer function cusolverDnSsyevj_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
```

13.3.66. cusolverDnDsyevj_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnDsyevj

```
integer function cusolverDnDsyevj_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
```

13.3.67. cusolverDnCheevj_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnCheevj

```
integer function cusolverDnCheevj_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
```

13.3.68. cusolverDnZheevj_buffersize

This function calculates the buffer sizes needed for the device workspace passed into cusolverDnZheevj

```
integer function cusolverDnZheevj_buffersize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    real(8), device, dimension(*) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
```

13.3.69. cusolverDnSsyevj

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnSsyevj(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
 type(cusolverDnSyevjInfo) :: params
```

13.3.70. cusolverDnDsyevj

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnDsyevj(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
```

13.3.71. cusolverDnCheevj

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnCheevj(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
```

13.3.72. cusolverDnZheevj

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix A.

```
integer function cusolverDnZheevj(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
```

13.3.73. cusolverDnSsyevjBatched_bufferSize

This function calculates the buffer size needed for the device workspace passed into cusolverDnSsyevjBatched.

```
integer function cusolverDnSsyevjBatched_bufferSize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.74. cusolverDnDsyevjBatched_bufferSize

This function calculates the buffer size needed for the device workspace passed into cusolverDnDsyevjBatched.

```
integer function cusolverDnDsyevjBatched_bufferSize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.75. cusolverDnCheevjBatched_bufferSize

This function calculates the buffer size needed for the device workspace passed into cusolverDnCheevjBatched.

```
integer function cusolverDnCheevjBatched_bufferSize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.76. cusolverDnZheevjBatched_bufferSize

This function calculates the buffer size needed for the device workspace passed into cusolverDnZheevjBatched.

```
integer function cusolverDnZheevjBatched_bufferSize(handle, &
    jobz, uplo, n, A, lda, W, lwork, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    integer(4), intent(out) :: lwork
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.77. cusolverDnSsyevjBatched

This function computes the eigenvalues and eigenvectors of symmetric or Hermitian nxn matrices.

```
integer function cusolverDnSsyevjBatched(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.78. cusolverDnDsyevjBatched

This function computes the eigenvalues and eigenvectors of symmetric or Hermitian nxn matrices.

```
integer function cusolverDnDsyevjBatched(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.79. cusolverDnCheevjBatched

This function computes the eigenvalues and eigenvectors of symmetric or Hermitian nxn matrices.

```
integer function cusolverDnCheevjBatched(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(4), device, dimension(lda,*) :: A
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.80. cusolverDnZheevjBatched

This function computes the eigenvalues and eigenvectors of symmetric or Hermitian nxn matrices.

```
integer function cusolverDnZheevjBatched(handle, &
    jobz, uplo, n, A, lda, W, workspace, lwork, devinfo, params, batchSize)
    type(cusolverDnHandle) :: handle
    integer(4) :: jobz, uplo
    integer(4) :: n, lda, lwork
    complex(8), device, dimension(lda,*) :: A
    real(8), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
    type(cusolverDnSyevjInfo) :: params
    integer(4) :: batchSize
```

13.3.81. cusolverDnSsygvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsygvd`

```
integer function cusolverDnSsygvd_buffersize(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    integer(4) :: lwork
```

13.3.82. cusolverDnDsygvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsygvd`

```
integer function cusolverDnDsygvd_buffersize(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    integer(4) :: lwork
```

13.3.83. cusolverDnChegvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnChegvd`

```
integer function cusolverDnChegvd_buffersize(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    integer(4) :: lwork
```

13.3.84. cusolverDnZhegvd_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZhegvd`

```
integer function cusolverDnZhegvd_buffersize(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    integer(4) :: lwork
```

13.3.85. cusolverDnSsygvd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnSsygvd(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb, lwork
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.86. cusolverDnDsygvd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnDsygvd(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb, lwork
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.87. cusolverDnChegvd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnChegvd(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb, lwork
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.88. cusolverDnZhegvd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnZhegvd(handle, &
    itype, jobz, uplo, n, A, lda, B, ldb, W, workspace, lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, uplo
    integer(4) :: n, lda, ldb, lwork
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.89. cusolverDnSsygvdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnSsygvdx`

```
integer function cusolverDnSsygvdx_buffersize(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu
    real(4) :: vl, vu
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.90. cusolverDnDsygvdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnDsygvdx`

```
integer function cusolverDnDsygvdx_buffersize(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu
    real(8) :: vl, vu
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.91. cusolverDnChegvdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnChegvdx`

```
integer function cusolverDnChegvdx_buffersize(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu
    real(4) :: vl, vu
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(ldb,*) :: B
    real(4), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.92. cusolverDnZhegvdx_buffersize

This function calculates the buffer sizes needed for the device workspace passed into `cusolverDnZhegvdx`

```
integer function cusolverDnZhegvdx_buffersize(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, lwork)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu
    real(8) :: vl, vu
    complex(8), device, dimension(lda,*) :: A
    complex(8), device, dimension(ldb,*) :: B
    real(8), device, dimension(n) :: W
    integer(4) :: meig, lwork
```

13.3.93. cusolverDnSsygvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnSsygvdx(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, workspace,
↪lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu, lwork
    real(4) :: vl, vu
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: meig
    real(4), device, dimension(n) :: W
    real(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.94. cusolverDnDsygvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnDsygvdx(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, workspace,
↪lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu, lwork
    real(8) :: vl, vu
    real(8), device, dimension(lda,*) :: A
    real(8), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: meig
    real(8), device, dimension(n) :: W
    real(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.95. cusolverDnChegvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnChegvdx(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, workspace,
↪lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu, lwork
    real(4) :: vl, vu
    complex(4), device, dimension(lda,*) :: A
    complex(4), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: meig
    real(4), device, dimension(n) :: W
    complex(4), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.3.96. cusolverDnZhegvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian nxn matrix pair (A,B).

```
integer function cusolverDnZhegvdx(handle, &
    itype, jobz, range, uplo, n, A, lda, B, ldb, vl, vu, il, iu, meig, W, workspace,
↪lwork, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: itype, jobz, range, uplo
    integer(4) :: n, lda, ldb, il, iu, lwork
    real(8) :: vl, vu
    complex(8), device, dimension(lda,*) :: A
```

(continues on next page)

(continued from previous page)

```
    complex(8), device, dimension(ldb,*) :: B
    integer(4), intent(out) :: meig
    real(8), device, dimension(n) :: W
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo
```

13.4. cusolverDn 64-bit API

This section describes the linear solver 64-bit API of cusolverDn, including Cholesky factorization, LU with partial pivoting, and QR factorization.

13.4.1. cusolverDnXpotrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverDnXpotrf`

The type and kind of the `A` matrix is determined by the dataTypeA argument. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXpotrf_buffersize(handle, &
    params, uplo, n, dataTypeA, A, lda, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: uplo
    integer(8) :: n, lda
    type(cudaDataType) :: dataTypeA, computeType
    real, device, dimension(lda,*) :: A
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.2. cusolverDnXpotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

The type and kind of the `A` matrix is determined by the dataTypeA argument. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXpotrf(handle, &
    params, uplo, n, dataTypeA, A, lda, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: uplo
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: n, lda
    type(cudaDataType) :: dataTypeA, computeType
    real, device, dimension(lda,*) :: A
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.3. cusolverDnXpotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverDnXpotrf`

The type and kind of the `A, B` matrices are determined by the dataTypeA and dataTypeB arguments, respectively. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXpotrs(handle, &
    params, uplo, n, nrhs, dataTypeA, A, lda, &
    dataTypeB, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: uplo
    integer(8) :: n, nrhs, lda, ldb
    type(cudaDataType) :: dataTypeA, dataTypeB
    real, device, dimension(lda,*) :: A
    real, device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.4.4. cusolverDnXgeqrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverDnXgeqrf`

The type and kind of the `A` matrix and `tau` vector is determined by the dataTypeA and dataTypeTau arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXgeqrf_buffersize(handle, &
    params, m, n, dataTypeA, A, lda, dataTypeTau, tau, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(8) :: m, n, lda
    type(cudaDataType) :: dataTypeA, dataTypeTau, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: tau
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.5. cusolverDnXgeqrf

This function computes the QR factorization of a general mxn matrix

The type and kind of the A matrix and tau vector is determined by the dataTypeA and dataTypeTau arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXgeqrf(handle, &
    params, m, n, dataTypeA, A, lda, dataTypeTau, tau, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(8) :: m, n, lda
    type(cudaDataType) :: dataTypeA, dataTypeTau, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: tau
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.6. cusolverDnXgetrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverDnXgetrf

The type and kind of the A matrix is determined by the dataTypeA argument. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXgetrf_buffersize(handle, &
    params, m, n, dataTypeA, A, lda, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(8) :: m, n, lda
    type(cudaDataType) :: dataTypeA, computeType
    real, device, dimension(lda,*) :: A
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.7. cusolverDnXgetrf

This function computes the LU factorization of a general mxn matrix

The type and kind of the A matrix is determined by the dataTypeA argument. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXgetrf(handle, &
    params, m, n, dataTypeA, A, lda, devipiv, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(8) :: n, m, lda
    type(cudaDataType) :: dataTypeA, computeType
    real, device, dimension(lda,*) :: A
    integer(8), device, dimension(*), intent(out) :: devipiv
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.8. cusolverDnXgetrs

This function solves the linear system of multiple right-hand sides applying a general nxn matrix which was LU-factored using `cusolverDnXgetrf`

The type and kind of the `A, B` matrices are determined by the dataTypeA and dataTypeB arguments, respectively. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXpotrs(handle, &
    params, trans, n, nrhs, dataTypeA, A, lda, devipiv, &
    dataTypeB, B, ldb, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: trans
    integer(8) :: n, nrhs, lda, ldb
    type(cudaDataType) :: dataTypeA, dataTypeB
    real, device, dimension(lda,*) :: A
    integer(8), device, dimension(*), intent(in) :: devipiv
    real, device, dimension(ldb,*) :: B
    integer(4), device, intent(out) :: devinfo
```

13.4.9. cusolverDnXsyevd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverDnXsyevd

The type and kind of the A matrix and W array is determined by the dataTypeA and dataTypeW arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXsyevd_buffersize(handle, &
    params, jobz, uplo, n, dataTypeA, A, lda, &
    dataTypeW, W, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, uplo
    integer(8) :: n, lda
    type(cudaDataType) :: dataTypeA, dataTypeW, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: W
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.10. cusolverDnXsyevd

This function computes the eigenvalues and eigenvectors of a symmetric (Hermitian) nxn matrix

The type and kind of the A matrix and W array is determined by the dataTypeA and dataTypeW arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXsyevd(handle, &
    params, jobz, uplo, n, dataTypeA, A, lda, &
    dataTypeW, W, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, uplo
    integer(8) :: n, lda
    type(cudaDataType) :: dataTypeA, dataTypeW, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: W
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.11. cusolverDnXsyevdx_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverDnXsyevdx

The type and kind of the A matrix and W array is determined by the dataTypeA and dataTypeW arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXsyevdx_buffersize(handle, &
    params, jobz, range, uplo, n, dataTypeA, A, lda, &
    vl, vu, il, iu, meig, &
    dataTypeW, W, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, range, uplo
    integer(8) :: n, lda, il, iu
    type(cudaDataType) :: dataTypeA, dataTypeW, computeType
    real, device, dimension(lda,*) :: A
    real :: vl, vu
    integer(8) :: meig
    real, device, dimension(*)     :: W
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.12. cusolverDnXsyevdx

This function computes all or some of the eigenvalues and optionally the eigenvectors of a symmetric (Hermitian) nxn matrix

The type and kind of the A matrix and W array is determined by the dataTypeA and dataTypeW arguments, respectively. The computeType is typically set to the same. Array sizes and dimensions, such as n, m, lda will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXsyevdx(handle, &
    params, jobz, range, uplo, n, dataTypeA, A, lda, &
    vl, vu, il, iu, meig, &
    dataTypeW, W, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, range, uplo
    integer(8) :: n, lda, il, iu
    type(cudaDataType) :: dataTypeA, dataTypeW, computeType
    real, device, dimension(lda,*) :: A
    real :: vl, vu
    integer(8) :: meig
    real, device, dimension(*)     :: W
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.13. cusolverDnXsytrs_bufferSize

This function calculates the buffer sizes needed to solve a system of linear equations using the generic API cusolverDnXsytrs function.

```
integer function cusolverDnXsytrs_bufferSize(handle, &
    uplo, n, nrhs, datatypeA, A, lda, &
    devipiv, datatypeB, B, ldb, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    type(cudaDataType) :: dataTypeA, dataTypeB
    integer(8) :: n, nrhs, lda, ldb
    real, device, dimension(lda,*) :: A
    integer(8), device :: devipiv(*)
    real, device, dimension(ldb,*) :: B
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.14. cusolverDnXsytrs

This function solves a system of linear equations using the generic API.

```
integer function cusolverDnXsytrs(handle, &
    uplo, n, nrhs, datatypeA, A, lda, &
    devipiv, datatypeB, B, ldb, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo
    type(cudaDataType) :: dataTypeA, dataTypeB
    integer(8) :: n, nrhs, lda, ldb
    real, device, dimension(lda,*) :: A
    integer(8), device :: devipiv(*)
    real, device, dimension(ldb,*) :: B
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.15. cusolverDnXtrtri_bufferSize

This function calculates the buffer sizes needed to compute the inverse of an upper or lower triangular matrix A using the generic API cusolverDnXtrtri function.

```
integer function cusolverDnXtrtri_bufferSize(handle, &
    uplo, diag, n, datatypeA, A, lda, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    type(cudaDataType) :: dataTypeA
    integer(8) :: n, lda
```

(continues on next page)

(continued from previous page)

```
    real, device, dimension(lda,*) :: A
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.16. cusolverDnXtrtri

This function computes the inverse of an upper or lower triangular matrix A using the generic API interface.

```
integer function cusolverDnXtrtri(handle, &
    uplo, diag, n, datatypeA, A, lda, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    integer(4) :: uplo, diag
    type(cudaDataType) :: dataTypeA
    integer(8) :: n, lda
    real, device, dimension(lda,*) :: A
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.17. cusolverDnXgesvd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverDnXgesvd`

The type and kind of the `A, U, and VT` matrices and the S array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXgesvd_buffersize(handle, &
    params, jobu, jobvt, m, n, dataTypeA, A, lda, &
    dataTypeS, S, dataTypeU, U, ldu, &
    dataTypeVT, VT, ldvt, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    character*1 :: jobu, jobvt
    integer(8) :: m, n, lda, ldu, ldvt
    type(cudaDataType) :: dataTypeA, dataTypeS, dataTypeU, dataTypeVT, &
    computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: S
    real, device, dimension(ldu,*) :: U
    real, device, dimension(ldvt,*) :: VT
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.18. cusolverDnXgesvd

This function computes the singular value decomposition (SVD) of an mxn matrix

The type and kind of the `A, U, and VT` matrices and the `S` array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXgesvd(handle, &
    params, jobu, jobvt, m, n, dataTypeA, A, lda, &
    dataTypeS, S, dataTypeU, U, ldu, &
    dataTypeVT, VT, ldvt, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    character*1 :: jobu, jobvt
    integer(8) :: m, n, lda, ldu, ldvt
    type(cudaDataType) :: dataTypeA, dataTypeS, dataTypeU, dataTypeVT, &
    computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: S
    real, device, dimension(ldu,*) :: U
    real, device, dimension(ldvt,*) :: VT
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.19. cusolverDnXgesvdp_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverDnXgesvdp`

The type and kind of the `A, U, and V` matrices and the `S` array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXgesvdp_buffersize(handle, &
    params, jobz, econ, m, n, dataTypeA, A, lda, &
    dataTypeS, S, dataTypeU, U, ldu, &
    dataTypeV, V, ldv, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, econ
    integer(8) :: m, n, lda, ldu, ldv
    type(cudaDataType) :: dataTypeA, dataTypeS, dataTypeU, dataTypeV, &
    computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: S
    real, device, dimension(ldu,*) :: U
    real, device, dimension(ldv,*) :: V
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.20. cusolverDnXgesvdp

This function computes the singular value decomposition (SVD) of an mxn matrix. This variation combines polar decompostion and `cusolverDnXsyevd` to compute the SVD.

The type and kind of the `A, U, and V` matrices and the S array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXgesvdp(handle, &
    params, jobz, econ, m, n, dataTypeA, A, lda, &
    dataTypeS, S, dataTypeU, U, ldu, &
    dataTypeV, V, ldv, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    integer(4) :: jobz, econ
    integer(8) :: m, n, lda, ldu, ldv
    type(cudaDataType) :: dataTypeA, dataTypeS, dataTypeU, dataTypeVT, &
    computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: S
    real, device, dimension(ldu,*) :: U
    real, device, dimension(ldv,*) :: V
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.4.21. cusolverDnXgesvdr_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverDnXgesvdr`

The type and kind of the `A, Urand, and Vrand` matrices and the `Srand` array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverDnXgesvdr_buffersize(handle, &
    params, jobu, jobvt, m, n, k, p, niters, dataTypeA, A, lda, &
    dataTypeSrand, Srand, dataTypeUrand, Urand, ldurand, &
    dataTypeVrand, Vrand, ldvrand, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    character*1 :: jobu, jobvt
    integer(8) :: m, n, k, p, niters, lda, ldurand, ldvrand
    type(cudaDataType) :: dataTypeA, dataTypeSrand, dataTypeUrand, &
      dataTypeVrand, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: Srand
```

(continues on next page)

(continued from previous page)

```
    real, device, dimension(ldurand,*) :: Urand
    real, device, dimension(ldvrand,*) :: Vrand
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.4.22. cusolverDnXgesvdr

This function computes the approximated rank-k singular value decomposition (SVD) of an mxn matrix.

The type and kind of the `A, Urand, and Vrand` matrices and the `Srand` array is determined by the associated dataType arguments. The computeType is typically set to the same. Array sizes and dimensions, such as `n, m, lda` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverDnXgesvdr(handle, &
    params, jobu, jobv, m, n, k, p, niters, dataTypeA, A, lda, &
    dataTypeSrand, Srand, dataTypeUrand, Urand, ldurand, &
    dataTypeVrand, Vrand, ldvrand, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverDnHandle) :: handle
    type(cusolverDnParams) :: params
    character*1 :: jobu, jobv
    integer(8) :: m, n, k, p, niters, lda, ldurand, ldvrand
    type(cudaDataType) :: dataTypeA, dataTypeSrand, dataTypeUrand, &
      dataTypeVrand, computeType
    real, device, dimension(lda,*) :: A
    real, device, dimension(*)     :: Srand
    real, device, dimension(ldurand,*) :: Urand
    real, device, dimension(ldvrand,*) :: Vrand
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5. cusolverMp API

This section describes the distributed, multi-processor linear solvers supported in cusolverMp, along with supporting functions to set up and initialize the library.

The cusolverMp module contains the following derived type definitions:

```
! Definitions from cusolverMp.h
integer, parameter :: CUSOLVERMP_VER_MAJOR = 0
integer, parameter :: CUSOLVERMP_VER_MINOR = 9
integer, parameter :: CUSOLVERMP_VER_PATCH = 1
integer, parameter :: CUSOLVERMP_VER_BUILD = 0
integer, parameter :: CUSOLVERMP_VERSION = &
       (CUSOLVERMP_VER_MAJOR * 10000 + CUSOLVERMP_VER_MINOR * 100 + CUSOLVERMP_VER_
↪PATCH)
```

```
! Types from cusolverMp.h
    TYPE cusolverMpHandle
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpHandle

    TYPE cusolverMpGrid
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpGrid

    TYPE cusolverMpMatrixDescriptor
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpMatrixDescriptor

    TYPE cusolverMpNewtonSchulzDescriptor
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpNewtonSchulzDescriptor

    TYPE cusolverMpPolarDescriptor
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpPolarDescriptor

    TYPE cusolverMpGesvdDescriptor
      TYPE(C_PTR) :: handle
    END TYPE cusolverMpGesvdDescriptor
```

```
enum, bind(c)
    enumerator :: CUSOLVERMP_GRID_MAPPING_COL_MAJOR = 0
    enumerator :: CUSOLVERMP_GRID_MAPPING_ROW_MAJOR = 1
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVERMP_NEWTON_SCHULZ_DESCRIPTOR_ATTRIBUTE_NORMALIZE = 0
    enumerator :: CUSOLVERMP_NEWTON_SCHULZ_DESCRIPTOR_ATTRIBUTE_REDUCE_VIA_COMPUTE_
↪TYPE = 1
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_REQUESTED_KSI = 0
    enumerator :: CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_A_NORM_FROBENIUS = 1
    enumerator :: CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_RCOND_ESTIMATE = 2
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_COMPUTE_RESIDUAL = 0
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_SHAPE = 1
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_A_NORM_FROBENIUS = 2
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_RCOND = 3
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_RESIDUAL_FROBENIUS_ESTIMATE =
↪4
    enumerator :: CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_NUM_SINGULAR_FOUND = 5
end enum
```

```
enum, bind(c)
    enumerator :: CUSOLVERMP_GESVD_OUTPUT_SHAPE_THIN = 0
    enumerator :: CUSOLVERMP_GESVD_OUTPUT_SHAPE_FULL = 1
end enum
```

13.5.1. cusolverMpCreate

This function initializes the cusolverMp library and creates a handle on the cusolverDn context. It must be called before other cuSolverMp API functions are invoked, but after a CUDA context and stream are created.

```
integer(4) function cusolverMpCreate(handle, deviceId, stream)
  type(cusolverMpHandle) :: handle
  integer(4) :: deviceId
  integer(cuda_stream_kind) :: stream
```

13.5.2. cusolverMpDestroy

This function releases CPU-side resources used by the cuSolverMp library.

```
integer(4) function cusolverMpDestroy(handle)
  type(cusolverMpHandle) :: handle
```

13.5.3. cusolverMpGetStream

This function gets the stream used by the cuSolverMp library to execute its routines.

```
integer(4) function cusolverMpGetStream(handle, stream)
  type(cusolverMpHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

13.5.4. cusolverMpSetStream

This function sets the stream used by the cuSolverMp library to execute its routines.

```
integer(4) function cusolverMpSetStream(handle, stream)
  type(cusolverMpHandle) :: handle
  integer(cuda_stream_kind) :: stream
```

13.5.5. cusolverMpGetVersion

This function gets the version of the cuSolverMp library at runtime. It has the value (CUSOLVERMP_VER_MAJOR * 1000 + CUSOLVERMP_VER_MINOR * 100 + CUSOLVERMP_VER_PATCH)

```
integer(4) function cusolverMpGetVersion(handle, version)
  type(cusolverMpHandle) :: handle
  integer(4) :: version
```

13.5.6. cusolverMpCreateDeviceGrid

This function initializes the grid data structure used in the cusolverMp library. It takes a handle, a communicator, and other information related to the data layout as inputs.

```
integer(4) function cusolverMpCreateDeviceGrid(handle, &
          grid, comm, numRowDevices, numColDevices, mapping)
  type(cusolverMpHandle) :: handle
  type(cusolverMpGrid) :: grid
  type(cal_comm) :: comm
  integer(4) :: numRowDevices, numColDevices
  integer(4) :: mapping ! enum above, usually column major in Fortran
```

13.5.7. cusolverMpDestroyGrid

This function destroys the grid data structure and frees the resources used in the cusolverMp library.

```
integer(4) function cusolverMpDestroyGrid(grid)
   type(cusolverMpGrid) :: grid
```

13.5.8. cusolverMpCreateMatrixDesc

This function initializes the matrix descriptor object used in the cusolverMp library. It takes the number of rows (M_A) and the number of columns (N_A) in the global array, along with the blocking factor over each dimension. RSRC_A and CSRC_A must currently be 0. LLD_A is the leading dimension of the local matrix, after blocking and distributing the matrix.

```
integer(4) function cusolverMpCreateMatrixDesc(descr, grid, &
          dataType, M_A, N_A, MB_A, NB_A, RSRC_A, CSRC_A, LLD_A)
  type(cusolverMpMatrixDescriptor)  :: descr
  type(cusolverMpGrid) :: grid
  type(cudaDataType)  :: dataType
  integer(8) :: M_A, N_A, MB_A, NB_A
  integer(4) :: RSRC_A, CSRC_A
  integer(8) :: LLD_A
```

13.5.9. cusolverMpDestroyMatrixDesc

This function destroys the matrix descriptor data structure and frees the resources used in the cusolverMp library.

```
integer(4) function cusolverMpDestroyMatrixDesc(descr)
   type(cusolverMpMatrixDescriptor) :: descr
```

13.5.10. cusolverMpNumROC

This utility function returns the number of rows or columns in the distributed matrix owned by the `iproc` process.

```
integer(4) function cusolverMpNumROC(N, NB, iproc, isrcproc, nprocs)
  integer(8), intent(in) :: N, NB
  integer(4), intent(in) :: iproc, isrcproc, nprocs
```

13.5.11. cusolverMpBufferRegister

This function registers a device buffer allocated with `ncclMemAlloc` for use with cusolverMp collective operations. Registering symmetric memory can improve collective performance. If registration is not supported, the caller should fall back to regular device memory.

```
integer(4) function cusolverMpBufferRegister(grid, ptr, size)
  type(cusolverMpGrid) :: grid
  type(c_devptr) :: ptr
  integer(8) :: size
```

13.5.12. cusolverMpBufferDeregister

This function deregisters a device buffer previously registered with `cusolverMpBufferRegister`. The buffer must be deregistered before it is freed.

```
integer(4) function cusolverMpBufferDeregister(grid, ptr)
  type(cusolverMpGrid) :: grid
  type(c_devptr) :: ptr
```

13.5.13. cusolverMpMalloc

This function allocates `size` bytes of device memory associated with the given grid, returning the address in `ptr`. Memory obtained this way is already known to the library, so it does not need to be passed to `cusolverMpBufferRegister`. Release it with `cusolverMpFree`.

```
integer(4) function cusolverMpMalloc(grid, ptr, size)
  type(cusolverMpGrid) :: grid
  type(c_devptr) :: ptr
  integer(8) :: size
```

13.5.14. cusolverMpFree

This function releases memory previously allocated by `cusolverMpMalloc` on the same grid.

```
integer(4) function cusolverMpFree(grid, ptr)
  type(cusolverMpGrid) :: grid
  type(c_devptr) :: ptr
```

13.5.15. cusolverMpMatrixScatterH2D

This function distributes an `M` by `N` matrix held contiguously on the host of rank `root`, with leading dimension `h_ldsrc`, into the block cyclic device array `A` described by `descrA`. It is the counterpart of `cusolverMpMatrixGatherD2H` and is typically used to load input data before a solve.

The type, kind, and rank of `A` and `h_src` are ignored; the element type is set by `descrA`. Array offsets and dimensions, such as `M, N, IA, JA`, will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer(4) function cusolverMpMatrixScatterH2D(handle, M, N, &
    A, IA, JA, descrA, root, h_src, h_ldsrc)
  type(cusolverMpHandle) :: handle
  integer(8) :: M, N, IA, JA
  real, device, dimension(*) :: A
  type(cusolverMpMatrixDescriptor) :: descrA
  integer(4) :: root
  real, dimension(*) :: h_src
  integer(8) :: h_ldsrc
```

13.5.16. cusolverMpMatrixGatherD2H

This function collects the `M` by `N` block cyclic device matrix `A` described by `descrA` into a contiguous host array `h_dst` on rank `root`, with leading dimension `h_lddst`. It is the counterpart of `cusolverMpMatrixScatterH2D` and is typically used to retrieve results for checking on the host.

The type, kind, and rank of `A` and `h_dst` are ignored; the element type is set by `descrA`. Array offsets and dimensions, such as `M, N, IA, JA`, will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer(4) function cusolverMpMatrixGatherD2H(handle, M, N, &
    A, IA, JA, descrA, root, h_dst, h_lddst)
  type(cusolverMpHandle) :: handle
  integer(8) :: M, N, IA, JA
  real, device, dimension(*) :: A
  type(cusolverMpMatrixDescriptor) :: descrA
  integer(4) :: root
  real, dimension(*) :: h_dst
  integer(8) :: h_lddst
```

13.5.17. cusolverMpGetrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpGetrf`.

The type, kind, and rank of the `A` matrix is ignored and has already been set by the `descrA` argument. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGetrf_buffersize(handle, &
    M, N, A, IA, JA, descrA, devipiv, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, IA, JA
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    integer(8), device, dimension(*) :: devipiv
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.18. cusolverMpGetrf

This function computes the LU factorization of a general mxn matrix

The type, kind, and rank of the `A` matrix is ignored and has already been set by the `descrA` argument. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverMpGetrf(handle, &
    M, N, A, IA, JA, descrA, devipiv, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, IA, JA
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    integer(8), device, dimension(*) :: devipiv
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.19. cusolverMpGetrs_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpGetrs`.

The type, kind, and rank of the `A, B` matrices are ignored and are set by the `descrA, descrB` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `N, NRHS` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGetrs_buffersize(handle, trans, &
    N, NRHS, A, IA, JA, descrA, devipiv, B, IB, JB, descrB, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: trans
    integer(8) :: N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    integer(8), device, dimension(*) :: devipiv
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.20. cusolverMpGetrs

This function solves the linear system of multiple right-hand sides applying a general nxn matrix which was LU-factored using `cusolverMpGetrf`

The type, kind, and rank of the `A, B` matrices are ignored and are set by the `descrA, descrB` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `N, NRHS` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGetrs(handle, trans, &
    N, NRHS, A, IA, JA, descrA, devipiv, B, IB, JB, descrB, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverMpHandle) :: handle
    integer(4) :: trans
    integer(8) :: N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    integer(8), device, dimension(*) :: devipiv
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5.21. cusolverMpPotrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverMpPotrf.

The type, kind, and rank of the A matrix is ignored and has already been set by the descrA argument. The computeType is typically set to the same type. Array sizes and dimensions, such as N will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPotrf_buffersize(handle, &
    uplo, N, A, IA, JA, descrA, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, IA, JA
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.22. cusolverMpPotrf

This function computes the Cholesky factorization of a Hermitian positive-definite matrix

The type, kind, and rank of the A matrix is ignored and has already been set by the descrA argument. The computeType is typically set to the same type. Array sizes and dimensions, such as N will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPotrf(handle, &
    uplo, N, A, IA, JA, descrA, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, IA, JA
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.23. cusolverMpPotrs_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverMpPotrs.

The type, kind, and rank of the A, B matrices are ignored and are set by the descrA, descrB arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as N, NRHS will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPotrs_buffersize(handle, uplo, &
    N, NRHS, A, IA, JA, descrA, B, IB, JB, descrB, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.24. cusolverMpPotrs

This function solves the system of linear equations resulting from the Cholesky factorization of a Hermitian positive-definite matrix using `cusolverMpPotrf`

The type, kind, and rank of the `A, B` matrices are ignored and are set by the `descrA, descrB` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `N, NRHS` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPotrs(handle, uplo, &
    N, NRHS, A, IA, JA, descrA, B, IB, JB, descrB, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.25. cusolverMpOrmqr_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpOrmqr`.

The type, kind, and rank of the `A, C` matrices is ignored and has already been set by the `descrA, descrC` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N, K` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpOrmqr_buffersize(handle, side, trans, &
    M, N, K, A, IA, JA, descrA, tau, C, IC, JC, descrC, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: side, trans
    integer(8) :: M, N, K, IA, JA, IC, JC
    real, device, dimension(*) :: A, tau, C
    type(cusolverMpMatrixDescriptor) :: descrA, descrC
```

(continues on next page)

(continued from previous page)

```
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.26. cusolverMpOrmqr

This function generates the unitary matrix Q from the QR factorization of an mxn matrix and overwrites the array C, based on the side and trans arguments.

The type, kind, and rank of the `A, C` matrices is ignored and has already been set by the `descrA, descrC` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N, K` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpOrmqr(handle, side, trans, &
    M, N, K, A, IA, JA, descrA, tau, C, IC, JC, descrC, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: side, trans
    integer(8) :: M, N, K, IA, JA, IC, JC
    real, device, dimension(*) :: A, tau, C
    type(cusolverMpMatrixDescriptor) :: descrA, descrC
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.27. cusolverMpOrmtr_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpOrmtr`.

The type, kind, and rank of the `A, tau, C` matrices is ignored and has already been set by the `descrA, descrC` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N, K` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpOrmtr_buffersize(handle, side, uplo, trans, &
    M, N, A, IA, JA, descrA, tau, C, IC, JC, descrC, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(8) :: M, N, IA, JA, IC, JC
    real, device, dimension(*) :: A, tau, C
    type(cusolverMpMatrixDescriptor) :: descrA, descrC
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.28. cusolverMpOrmtr

This function generates the unitary matrix Q formed by a sequence of elementary reflection vectors and overwrites the array C, based on the side and trans arguments.

The type, kind, and rank of the `A, C` matrices is ignored and has already been set by the `descrA, descrC` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N, K` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpOrmtr(handle, side, uplo, trans, &
    M, N, A, IA, JA, descrA, tau, C, IC, JC, descrC, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: side, uplo, trans
    integer(8) :: M, N, IA, JA, IC, JC
    real, device, dimension(*) :: A, tau, C
    type(cusolverMpMatrixDescriptor) :: descrA, descrC
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.29. cusolverMpGels_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpGels`.

The type, kind, and rank of the `A, B` matrices are ignored and are set by the `descrA, descrB` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `N, NRHS` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGels_buffersize(handle, trans, &
    M, N, NRHS, A, IA, JA, descrA, B, IB, JB, descrB, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: trans
    integer(8) :: M, N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.30. cusolverMpGels

This function solves overdetermined or underdetermined linear systems of the MxN matrix A, or its transpose, using QR or LQ factorization.

The type, kind, and rank of the `A, B` matrices are ignored and are set by the `descrA, descrB` arguments. The computeType is typically set to the same type. Array sizes and dimensions, such as `N, NRHS` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGels(handle, trans, &
    M, N, NRHS, A, IA, JA, descrA, B, IB, JB, descrB, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: trans
    integer(8) :: M, N, NRHS, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.31. cusolverMpStedc_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpStedc`.

The type, kind, and rank of the `D, E, Q` matrices are ignored and are already set by the `descrQ` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IQ, JQ` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpStedc_buffersize(handle, compz, &
    N, D, E, Q, IQ, JQ, descrQ, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    character :: compz
    integer(8) :: N, IQ, JQ
    real, device, dimension(*) :: D, E, Q
    type(cusolverMpMatrixDescriptor) :: descrQ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(4) :: iwork(*)
```

13.5.32. cusolverMpStedc

This function computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

The type, kind, and rank of the `D, E, Q` matrices are ignored and are already set by the `descrQ` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IQ, JQ` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpStedc(handle, compz, &
    N, D, E, Q, IQ, JQ, descrQ, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    character :: compz
    integer(8) :: N, IQ, JQ
    real, device, dimension(*) :: D, E, Q
    type(cusolverMpMatrixDescriptor) :: descrQ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.33. cusolverMpGeqrf_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpGeqrf`.

The type, kind, and rank of the `A` matrix is ignored and has already been set by the `descrA` argument. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGeqrf_buffersize(handle, &
    M, N, A, IA, JA, descrA, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, IA, JA
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.34. cusolverMpGeqrf

This function computes a QR factorization of an MxN matrix.

The type, kind, and rank of the A matrix is ignored and has already been set by the descrA argument. The computeType is typically set to the same type. Array sizes and dimensions, such as M, N, K will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGeqrf(handle, &
    M, N, A, IA, JA, descrA, tau, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, IA, JA
    real, device, dimension(*) :: A, tau
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.35. cusolverMpOrgqr_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverMpOrgqr.

The type, kind, and rank of the A matrix and tau vector are ignored and have already been set by the descrA argument. The computeType is typically set to the same type. Array sizes and dimensions, such as M, N, K will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpOrgqr_buffersize(handle, &
    M, N, K, A, IA, JA, descrA, tau, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, K, IA, JA
    real, device, dimension(*) :: A, tau
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.36. cusolverMpOrgqr

This function generates an MxN real matrix Q with orthonormal columns from the result of cusolverMpGeqrf. Q is defined as the first N columns of the product of K elementary reflectors of order M: Q = H(1) H(2) ... H(K).

The type, kind, and rank of the A matrix and tau vector are ignored and have already been set by the descrA argument. The computeType is typically set to the same type. Array sizes and dimensions, such as M, N, K will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverMpOrgqr(handle, &
    M, N, K, A, IA, JA, descrA, tau, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverMpHandle) :: handle
    integer(8) :: M, N, K, IA, JA
    real, device, dimension(*) :: A, tau
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5.37. cusolverMpSytrd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpSytrd`.

The type, kind, and rank of the `A, D, E` matrix and vectors are ignored and are already set by the `descrA` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSytrd_buffersize(handle, uplo, &
    N, A, IA, JA, descrA, D, E, tau, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, IA, JA
    real, device, dimension(*) :: A, D, E, tau
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.38. cusolverMpSytrd

This function reduces a symmetric (Hermitian) NxN matrix to symmetric tridiagonal form.

The type, kind, and rank of the `A, D, E` matrix and vectors are ignored and are already set by the `descrA` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSytrd(handle, uplo, &
    N, A, IA, JA, descrA, D, E, tau, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: N, IA, JA
    real, device, dimension(*) :: A, D, E, tau
    type(cusolverMpMatrixDescriptor) :: descrA
    type(cudaDataType) :: computeType
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.39. cusolverMpSyevd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpSyevd`.

The type, kind, and rank of the `A, D, Q` matrices are ignored and are already set by the `descrA, descrQ` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSyevd_buffersize(handle, compz, &
    uplo, N, A, IA, JA, descrA, D, Q, IQ, JQ, descrQ, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    character :: compz
    integer(4) :: uplo
    integer(8) :: N, IA, JA, IQ, JQ
    real, device, dimension(*) :: A, D, Q
    type(cusolverMpMatrixDescriptor) :: descrA, descrQ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.40. cusolverMpSyevd

This function computes the eigenvalues and eigenvectors of a symmetric (Hermitian) nxn matrix

The type, kind, and rank of the `A, D, Q` matrices are ignored and are already set by the `descrA, descrQ` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSyevd(handle, compz, &
    uplo, N, A, IA, JA, descrA, D, Q, IQ, JQ, descrQ, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverMpHandle) :: handle
    character :: compz
    integer(4) :: uplo
    integer(8) :: N, IA, JA, IQ, JQ
    real, device, dimension(*) :: A, D, Q
    type(cusolverMpMatrixDescriptor) :: descrA, descrQ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5.41. cusolverMpSygst_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverMpSygst.

The types for the A, B matrices are set by the descrA, descrB arguments and not passed in to compute the buffer requirements. The computeType is typically set to the same type. Array offsets and dimensions, such as M, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSygst_buffersize(handle, ibtype, uplo, &
    M, IA, JA, descrA, IB, JB, descrB, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: ibtype, uplo
    integer(8) :: M, IA, JA, IB, JB
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.42. cusolverMpSygst

This function reduces a symmetric-definite generalized eigenproblem to standard form.

The type, kind, and rank of the A, B matrices are ignored and are already set by the descrA, descrB argument. The computeType is typically set to the same type. Array offsets and dimensions, such as M, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSygst(handle, ibtype, uplo, &
    M, A, IA, JA, descrA, B, IB, JB, descrB, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: ibtype, uplo
    integer(8) :: M, IA, JA, IB, JB
    real, device, dimension(*) :: A, B
    type(cusolverMpMatrixDescriptor) :: descrA, descrB
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.43. cusolverMpSygvd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into cusolverMpSygvd.

The types for the A, B, Z matrices are set by the descrA, descrB, descrZ arguments and not passed in to compute the buffer requirements. The computeType is typically set to the same type. Array offsets and dimensions, such as M, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSygvd_buffersize(handle, ibtype, jobz, &
    uplo, M, IA, JA, descrA, IB, JB, descrB, IZ, JZ, descrZ, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    integer(4) :: ibtype, jobz, uplo
    integer(8) :: M, IA, JA, IB, JB, IZ, JZ
    type(cusolverMpMatrixDescriptor) :: descrA, descrB, descrZ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.44. cusolverMpSygvd

This function computes all eigenvalues and eigenvectors of a symmetric (Hermitian) generalized eigen problem.

The type, kind, and rank of the A, B, Z matrices are ignored and are already set by the descrA, descrB, descrZ argument. The computeType is typically set to the same type. Array offsets and dimensions, such as M, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpSygvd(handle, ibtype, jobz, &
    uplo, M, A, IA, JA, descrA, B, IB, JB, descrB, &
    W, Z, IZ, JZ, descrZ, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
    type(cusolverMpHandle) :: handle
    integer(4) :: ibtype, jobz, uplo
    integer(8) :: M, IA, JA, IB, JB,
    real, device, dimension(*) :: A, B, W, Z
    type(cusolverMpMatrixDescriptor) :: descrA, descrB, descrZ
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.45. cusolverMpLaset

This function initializes a distributed MxN matrix A. The off-diagonal elements are set to `alpha` and the diagonal elements are set to `beta`. The `uplo` argument controls whether the upper triangular, lower triangular, or full matrix is set.

The type, kind, and rank of the `A` matrix and the scalars `alpha, beta` are ignored and have already been set by the `descrA` argument. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpLaset(handle, uplo, &
    M, N, alpha, beta, A, IA, JA, descrA, devinfo)
    type(cusolverMpHandle) :: handle
    integer(4) :: uplo
    integer(8) :: M, N, IA, JA
    real :: alpha, beta
    real, device, dimension(*) :: A
    type(cusolverMpMatrixDescriptor) :: descrA
    integer(4), device, intent(out) :: devinfo
```

13.5.46. cusolverMpNewtonSchulzDescriptorCreate

This function creates a Newton-Schulz descriptor object used to configure the Newton-Schulz orthogonalization solver.

```
integer(4) function cusolverMpNewtonSchulzDescriptorCreate(nsDesc)
  type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
```

13.5.47. cusolverMpNewtonSchulzDescriptorDestroy

This function destroys a Newton-Schulz descriptor object and frees associated resources.

```
integer(4) function cusolverMpNewtonSchulzDescriptorDestroy(nsDesc)
  type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
```

13.5.48. cusolverMpNewtonSchulzDescriptorSetAttribute

This function sets an attribute on the Newton-Schulz descriptor. The `attr` argument is one of the `CUSOLVERMP_NEWTON_SCHULZ_DESCRIPTOR_ATTRIBUTE_*` enumerators. The `buf` argument is a host buffer containing the attribute value, and `sizeInBytes` specifies its size.

```
integer(4) function cusolverMpNewtonSchulzDescriptorSetAttribute( &
    nsDesc, attr, buf, sizeInBytes)
  type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
```

13.5.49. cusolverMpNewtonSchulzDescriptorGetAttribute

This function gets an attribute from the Newton-Schulz descriptor. The `attr` argument is one of the `CUSOLVERMP_NEWTON_SCHULZ_DESCRIPTOR_ATTRIBUTE_*` enumerators. The value is written into `buf` and the number of bytes written is returned in `sizeInBytesWritten`.

```
integer(4) function cusolverMpNewtonSchulzDescriptorGetAttribute( &
    nsDesc, attr, buf, sizeInBytes, sizeInBytesWritten)
  type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
  integer(8) :: sizeInBytesWritten
```

13.5.50. cusolverMpNewtonSchulz_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpNewtonSchulz`.

The type, kind, and rank of the `X` matrix and `h_coeffs` vector are ignored and have already been set by the `descX` argument. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpNewtonSchulz_buffersize(handle, nsDesc, &
    M, N, X, IX, JX, descX, numberOfIterations, h_coeffs, &
    computeType, workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
    integer(8) :: M, N, IX, JX, numberOfIterations
    real, device, dimension(*) :: X
    type(cusolverMpMatrixDescriptor) :: descX
    real, dimension(*) :: h_coeffs
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.51. cusolverMpNewtonSchulz

This function performs approximate orthogonalization of an MxN matrix X using the Newton-Schulz iteration. The iteration is parameterized by `numberOfIterations` quintic steps and a coefficient vector `h_coeffs` of length `3 * numberOfIterations`. On output, X is overwritten with the orthogonalized matrix.

The type, kind, and rank of the `X` matrix and `h_coeffs` vector are ignored and have already been set by the `descX` argument. The computeType is typically set to the same type. Array sizes and dimensions, such as `M, N` will be promoted to integer(8) by the compiler, so other integer kinds may be used. The workspaces can be any type and kind so long as the adequate amount of space in bytes is available.

```
integer function cusolverMpNewtonSchulz(handle, nsDesc, &
    M, N, X, IX, JX, descX, numberOfIterations, h_coeffs, &
    computeType, bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, info)
```

(continues on next page)

(continued from previous page)

```
    type(cusolverMpHandle) :: handle
    type(cusolverMpNewtonSchulzDescriptor) :: nsDesc
    integer(8) :: M, N, IX, JX, numberOfIterations
    real, device, dimension(*) :: X
    type(cusolverMpMatrixDescriptor) :: descX
    real, dimension(*) :: h_coeffs
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: info
```

13.5.52. cusolverMpPolarDescriptorCreate

This function creates a descriptor used to configure `cusolverMpPolar` and to read back its diagnostics. Release it with `cusolverMpPolarDescriptorDestroy`.

```
integer(4) function cusolverMpPolarDescriptorCreate(polarDesc)
  type(cusolverMpPolarDescriptor) :: polarDesc
```

13.5.53. cusolverMpPolarDescriptorDestroy

This function destroys a polar decomposition descriptor and releases its resources.

```
integer(4) function cusolverMpPolarDescriptorDestroy(polarDesc)
  type(cusolverMpPolarDescriptor) :: polarDesc
```

13.5.54. cusolverMpPolarDescriptorSetAttribute

This function sets an attribute on the polar decomposition descriptor. The `attr` argument is one of the `CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_*` enumerators, `buf` is a host buffer holding the value to set, and `sizeInBytes` is its size. For `CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_REQUESTED_KSI`, `buf` holds a double precision value: zero or less, the default, requests no perturbation of the input, and only a value greater than zero requests a perturbed problem.

```
integer(4) function cusolverMpPolarDescriptorSetAttribute( &
    polarDesc, attr, buf, sizeInBytes)
  type(cusolverMpPolarDescriptor) :: polarDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
```

13.5.55. cusolverMpPolarDescriptorGetAttribute

This function gets an attribute from the polar decomposition descriptor. The `attr` argument is one of the `CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_*` enumerators. The value is written into `buf` and the number of bytes written is returned in `sizeInBytesWritten`. For `CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_A_NORM_FROBENIUS` and `CUSOLVERMP_POLAR_DESCRIPTOR_ATTRIBUTE_RCOND_ESTIMATE`, `buf` receives a double precision value, available after the solve.

```
integer(4) function cusolverMpPolarDescriptorGetAttribute( &
    polarDesc, attr, buf, sizeInBytes, sizeInBytesWritten)
  type(cusolverMpPolarDescriptor) :: polarDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
  integer(8) :: sizeInBytesWritten
```

13.5.56. cusolverMpPolar_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpPolar`.

The type, kind, and rank of the `A, H` matrices are ignored and are already set by the `descrA, descrH` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `M, N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPolar_buffersize(handle, polarDesc, &
    uplo, M, N, A, IA, JA, descrA, H, IH, JH, descrH, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    type(cusolverMpPolarDescriptor) :: polarDesc
    integer(4) :: uplo
    integer(8) :: M, N, IA, JA, IH, JH
    real, device, dimension(*) :: A, H
    type(cusolverMpMatrixDescriptor) :: descrA, descrH
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.57. cusolverMpPolar

This function computes the polar decomposition `A = Up*H` of an `M` by `N` matrix with `M >= N`, using the QDWH algorithm. `H` is `N` by `N` symmetric (Hermitian) positive semi-definite. `Up` has orthonormal columns for a full rank input; for a rank deficient input that is not perturbed, orthonormality is not guaranteed. Set `uplo` to `CUBLAS_FILL_MODE_FULL` when `A` is a general dense matrix.

Computing `H` is optional. Passing a null `descrH`, that is a `cusolverMpMatrixDescriptor` whose handle component is `C_NULL_PTR`, skips it and returns only `Up`.

On return `A` has been overwritten with `Up` and is not recoverable, and `H` holds the symmetric factor when it was requested. Keep a copy of the input if it is needed afterwards.

The type, kind, and rank of the A, H matrices are ignored and are already set by the descrA, descrH argument. The computeType is typically set to the same type. Array offsets and dimensions, such as M, N, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpPolar(handle, polarDesc, &
    uplo, M, N, A, IA, JA, descrA, H, IH, JH, descrH, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverMpHandle) :: handle
    type(cusolverMpPolarDescriptor) :: polarDesc
    integer(4) :: uplo
    integer(8) :: M, N, IA, JA, IH, JH
    real, device, dimension(*) :: A, H
    type(cusolverMpMatrixDescriptor) :: descrA, descrH
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5.58. cusolverMpGesvdDescriptorCreate

This function creates a descriptor used to configure cusolverMpGesvd and to read back its diagnostics. Release it with cusolverMpGesvdDescriptorDestroy. The descriptor is optional for the solve itself, but it is the only way to retrieve the residual, condition and singular value count reported below.

```
integer(4) function cusolverMpGesvdDescriptorCreate(gesvdDesc)
  type(cusolverMpGesvdDescriptor) :: gesvdDesc
```

13.5.59. cusolverMpGesvdDescriptorDestroy

This function destroys a singular value decomposition descriptor and releases its resources.

```
integer(4) function cusolverMpGesvdDescriptorDestroy(gesvdDesc)
  type(cusolverMpGesvdDescriptor) :: gesvdDesc
```

13.5.60. cusolverMpGesvdDescriptorSetAttribute

This function sets an attribute on the singular value decomposition descriptor. The attr argument is one of the CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_* enumerators, buf is a host buffer holding the value to set, and sizeInBytes is its size. For CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_COMPUTE_RESIDUAL, buf holds a default integer, and setting it to 1 makes the residual estimate available afterwards. For CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_SHAPE, buf holds one of the CUSOLVERMP_GESVD_OUTPUT_SHAPE_* enumerators.

```
integer(4) function cusolverMpGesvdDescriptorSetAttribute( &
    gesvdDesc, attr, buf, sizeInBytes)
```

(continues on next page)

(continued from previous page)

```
  type(cusolverMpGesvdDescriptor) :: gesvdDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
```

13.5.61. cusolverMpGesvdDescriptorGetAttribute

This function gets an attribute from the singular value decomposition descriptor. The `attr` argument is one of the `CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_*` enumerators. The value is written into `buf` and the number of bytes written is returned in `sizeInBytesWritten`. For `CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_A_NORM_FROBENIUS`, `CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_RCOND` and `CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_RESIDUAL` `buf` receives a double precision value. For `CUSOLVERMP_GESVD_DESCRIPTOR_ATTRIBUTE_NUM_SINGULAR_FOUND`, `buf` receives an integer(8).

```
integer(4) function cusolverMpGesvdDescriptorGetAttribute( &
    gesvdDesc, attr, buf, sizeInBytes, sizeInBytesWritten)
  type(cusolverMpGesvdDescriptor) :: gesvdDesc
  integer(4) :: attr
  real, dimension(*) :: buf
  integer(8) :: sizeInBytes
  integer(8) :: sizeInBytesWritten
```

13.5.62. cusolverMpGesvd_buffersize

This function calculates the buffer sizes needed for the host and device workspaces passed into `cusolverMpGesvd`.

The type, kind, and rank of the `A, S, U, VT` matrices and vector are ignored and are already set by the `descrA, descrU, descrVT` argument. The computeType is typically set to the same type. Array offsets and dimensions, such as `M, N, IA, JA` will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGesvd_buffersize(handle, gesvdDesc, &
    jobu, jobvt, M, N, A, IA, JA, descrA, S, U, IU, JU, descrU, &
    VT, IVT, JVT, descrVT, computeType, &
    workspaceInBytesOnDevice, workspaceInBytesOnHost)
    type(cusolverMpHandle) :: handle
    type(cusolverMpGesvdDescriptor) :: gesvdDesc
    integer(4) :: jobu, jobvt
    integer(8) :: M, N, IA, JA, IU, JU, IVT, JVT
    real, device, dimension(*) :: A, S, U, VT
    type(cusolverMpMatrixDescriptor) :: descrA, descrU, descrVT
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
```

13.5.63. cusolverMpGesvd

This function computes the singular value decomposition of an M by N matrix: A = U*diag(S)*V**T for real data, and A = U*diag(S)*V**H for complex data, where V**H is the conjugate transpose. The output argument is named VT in both cases. jobu and jobvt are cusolverEigMode_t values, CUSOLVER_EIG_MODE_VECTOR or CUSOLVER_EIG_MODE_NOVECTOR, which are provided by the cusolver_common module. With the default thin output shape, U is M by K, VT is K by N and S has K elements, where K is the smaller of M and N.

On return A has been overwritten with intermediate state and is not recoverable. Keep a copy of the input if it is needed afterwards, for instance to compute a residual.

The type, kind, and rank of the A, S, U, VT matrices and vector are ignored and are already set by the descrA, descrU, descrVT argument. The computeType is typically set to the same type. Array offsets and dimensions, such as M, N, IA, JA will be promoted to integer(8) by the compiler, so other integer kinds may be used.

```
integer function cusolverMpGesvd(handle, gesvdDesc, &
    jobu, jobvt, M, N, A, IA, JA, descrA, S, U, IU, JU, descrU, &
    VT, IVT, JVT, descrVT, computeType, &
    bufferOnDevice, workspaceInBytesOnDevice, &
    bufferOnHost, workspaceInBytesOnHost, devinfo)
    type(cusolverMpHandle) :: handle
    type(cusolverMpGesvdDescriptor) :: gesvdDesc
    integer(4) :: jobu, jobvt
    integer(8) :: M, N, IA, JA, IU, JU, IVT, JVT
    real, device, dimension(*) :: A, S, U, VT
    type(cusolverMpMatrixDescriptor) :: descrA, descrU, descrVT
    type(cudaDataType) :: computeType
    integer(8) :: workspaceInBytesOnDevice, workspaceInBytesOnHost
    integer(1), device :: bufferOnDevice(workspaceInBytesOnDevice)
    integer(1)         :: bufferOnHost(workspaceInBytesOnHost)
    integer(4), device, intent(out) :: devinfo
```

13.5.64. cusolverMpSetMathMode

This function sets the math mode for the cusolverMp library handle. The mode argument is a cusolverMathMode_t enumerator value. Available when CUSOLVER_VERSION >= 12000.

```
integer(4) function cusolverMpSetMathMode(handle, mode)
  type(cusolverMpHandle) :: handle
  integer(4) :: mode
```

13.5.65. cusolverMpGetMathMode

This function gets the current math mode from the cusolverMp library handle. Available when `CUSOLVER_VERSION >= 12000`.

```
integer(4) function cusolverMpGetMathMode(handle, mode)
  type(cusolverMpHandle) :: handle
  integer(4), intent(out) :: mode
```

13.5.66. cusolverMpSetEmulationStrategy

This function sets the emulation strategy for the cusolverMp library handle. The `strategy` argument is a `cublasEmulationStrategy_t` enumerator value. Available when `CUSOLVER_VERSION >= 12000`.

```
integer(4) function cusolverMpSetEmulationStrategy(handle, strategy)
  type(cusolverMpHandle) :: handle
  integer(4) :: strategy
```

13.5.67. cusolverMpGetEmulationStrategy

This function gets the current emulation strategy from the cusolverMp library handle. Available when `CUSOLVER_VERSION >= 12000`.

```
integer(4) function cusolverMpGetEmulationStrategy(handle, strategy)
  type(cusolverMpHandle) :: handle
  integer(4), intent(out) :: strategy
```

13.5.68. cusolverMpLoggerSetFile

This function specifies the Fortran unit to be used as the cusolverMp logfile.

```
integer(4) function cusolverMpLoggerSetFile(unit)
  integer :: unit
```

13.5.69. cusolverMpLoggerOpenFile

This function specifies a Fortran character string to be opened and used as the cusolverMp logfile.

```
integer(4) function cusolverMpLoggerOpenFile(logFile)
  character*(*) :: logFile
```

13.5.70. cusolverMpLoggerSetLevel

This function specifies the cusolverMp logging level.

```
integer(4) function cusolverMpLoggerSetLevel(level)
  integer :: level
```

13.5.71. cusolverMpLoggerSetMask

This function specifies the cusolverMp logging mask.

```
integer(4) function cusolverMpLoggerSetMask(mask)
  integer :: mask
```

13.5.72. cusolverMpLoggerForceDisable

This function disables cusolverMp logging.

```
integer(4) function cusolverMpLoggerForceDisable()
```

13.5.73. cusolverMpLoggerSetCallback

This function installs a callback which receives cusolverMp log messages. Build the argument with `c_funloc` applied to a `bind(C)` subroutine, and pass `c_null_funptr` to remove a previously installed callback.

The callback takes the log level, the function name and the message. The two names arrive as C pointers to null terminated strings; use `c_f_pointer` to read them.

```
integer(4) function cusolverMpLoggerSetCallback(callback)
  type(c_funptr) :: callback
```

The callback itself has the following interface.

```
subroutine logger_callback(logLevel, functionName, message) bind(C)
  integer(c_int), value :: logLevel
  type(c_ptr), value :: functionName, message
```

13.6. NVLAMATH Runtime Library

This section describes the NVLAmath runtime library, which enables offloading calls to standard LAPACK subroutines to the GPU by redirecting them to a wrapper around cuSOLVER functions.

Functionality and performance wise, this library provides no additions to what is documented in the previous sections of this chapter. The most common use cases for NVLAmath we see are:

- Drop-in acceleration. When running in a unified memory environment, NVLAmath allows recompile-and-run to offload LAPACK solvers to the GPU.
- Quick-porting functionality which may or may not be performance critical, but where the data already resides on the GPU from other sections of the application, via CUDA, OpenACC, or OpenMP. Access to a GPU-enabled function avoids costly data movement back and forth between the GPU and CPU.
- Keeping traditional code integrity. NVLAmath enables porting to the GPU with minimal code changes, for instance enabling 32-bit and 64-bit integers in the same source, and calling solver library entry points which are available on all targets.

The NVLAmath wrappers are enabled through the Fortran nvlamath module, which overloads the LAPACK subroutine names. There are two ways users can insert the module into their Fortran subprogram units. The first is non-invasive: add the compiler option `-gpu=nvlamath` to the compile lines on the files containing LAPACK calls you wish to move to the GPU. The second way is more traditional, just add the `use nvlamath` statement to the subroutines or functions that contain those LAPACK calls. If you use neither of these methods, the LAPACK calls will likely be resolved by the CPU LAPACK library, either the one we ship or the one you decide to use on your link line. In some cases, the NVLAmath wrappers might even call into that CPU library as well. In other words, there is no tampering with the CPU LAPACK symbol (library entry point) names.

To link the NVLAmath wrapper library, add `-cudalib=nvlamath` to your link line. To use the 64-bit integer version of the library, either compile and link with the `-i8` option, or add `-cudalib=nvlamath_ilp64` to your link line.

13.6.1. NVLAMATH Automatic Drop-In Acceleration

In unified memory environments, the simplest method to move standard LAPACK calls to run on the GPU is through what we call automatic drop-in acceleration. This currently requires a GPU and OS which supports CUDA managed memory, and that the arrays that are passed to the library functions are dynamically allocated.

Here is an example test program which calls dgetrf() and dgetrs():

```
program testdgetrf
integer, parameter :: M = 1000, N = M, NRHS=2
real(8), allocatable :: A(:,:), B(:,:)
integer, allocatable :: IPIV(:)
real(8), parameter :: eps = 1.0d-10
!
allocate(A(M,N),B(M,NRHS),IPIV(M))
call random_number(A)
do i = 1, M
    A(i,i) = A(i,i) * 10.0d0
    B(i,1) = sum(A(i,:))
```

(continues on next page)

(continued from previous page)

```
    B(i,2) = B(i,1) * 2.0d0
end do
!
lda = M; ldb = M
call dgetrf(M,N,A,lda,IPIV,info1)
call dgetrs('n',n,NRHS,A,lda,IPIV,B,ldb,info2)
!
if ((info1.ne.0) .or. (info2.ne.0)) then
    print *,"test FAILED"
else if (any(abs(B(:,1)-1.0d0) .gt. eps) .or. &
         any(abs(B(:,2)-2.0d0) .gt. eps)) then
    print *,"test FAILED"
else
    print *,"test PASSED"
end if
end
```

To compile and link this program for CPU execution using nvfortran, the simplest option would be `nvfortran testdgetrf.f90 -llapack -lblas`. To enable acceleration on the GPU, the simplest option would be `nvfortran -stdpar -gpu=nvlamath testdgetrf.f90 -cudalib=nvlamath`. Here, `-stdpar` is one of the available methods to instruct the compiler and runtime to treat allocatable arrays as CUDA managed data. The `-gpu=nvlamath` option instructs the compiler to insert the statement `use nvhpc_nvlamath` into the program unit and then the two LAPACK calls are redirected to the wrappers. Finally, the `-cudalib=nvlamath` linker option pulls in the NVLAmath wrapper runtime library.

13.6.2. NVLAMATH Usage From CUDA Fortran

From CUDA Fortran, usage of NVLAmath is very straight-forward. The interfaces in the `nvhpc_nvlamath` module are similar in nature to all the other Fortran modules in this document. The library function device dummy arguments can be matched to device or managed actual arguments, or a combination of both, as used here.

This example code is written in CUDA Fortran. Note that all CUDA Fortran additions are behind the `!@cuf` sentinel, or within a CUF kernel, which still allows this file to be compiled with any Fortran compiler.

```
program testdgetrf
!@cuf use nvhpc_nvlamath
!@cuf use cutensorex
integer, parameter :: M = 1000, N = M, NRHS=2
real(8), allocatable :: A(:,:), B(:,:)
integer, allocatable :: IPIV(:)
!@cuf attributes(device) :: A, IPIV
!@cuf attributes(managed) :: B
real(8), parameter :: eps = 1.0d-10
!
allocate(A(M,N),B(M,NRHS),IPIV(M))
call random_number(A)
!$cuf kernel do(1)<<<*,*>>>
do i = 1, M
    A(i,i) = A(i,i) * 10.0d0
    B(i,1) = sum(A(i,:))
```

(continues on next page)

(continued from previous page)

```
    B(i,2) = B(i,1) * 2.0d0
end do
!
lda = M; ldb = M
call dgetrf(M,N,A,lda,IPIV,info1)
call dgetrs('n',n,NRHS,A,lda,IPIV,B,ldb,info2)
!
if ((info1.ne.0) .or. (info2.ne.0)) then
    print *,"test FAILED"
else if (any(abs(B(:,1)-1.0d0) .gt. eps) .or. &
         any(abs(B(:,2)-2.0d0) .gt. eps)) then
    print *,"test FAILED"
else
    print *,"test PASSED"
end if
end
```

Here we use the `nvhpc_nvlamath` module explicitly, and we also use the `cutensorex` module, discussed elsewhere in this document, to generate random numbers on the GPU. Assuming this file has a .cuf extension, the simplest compile and link line to use with this file is `nvfortran testdgetrf.cuf -cudalib=curand,nvlamath`.

13.6.3. NVLAMATH Usage From OpenACC

From OpenACC, the LAPACK functions are treated just like CUBLAS or CUSOLVER functions. Use the `host_data use_device` directive to pass device pointers to the functions.

```
program testdgetrf
integer, parameter :: M = 1000, N = M, NRHS=2
real(8) :: A(M,N), B(M,NRHS)
integer :: IPIV(M)
real(8), parameter :: eps = 1.0d-10
!
call random_number(A)
!$acc data copyin(A), copyout(B), create(IPIV)
!$acc parallel loop
do i = 1, M
  A(i,i) = A(i,i) * 10.0d0
  B(i,1) = sum(A(i,:))
  B(i,2) = B(i,1) * 2.0d0
end do
!
lda = M; ldb = M
!$acc host_data use_device(A, IPIV, B)
call dgetrf(M,N,A,lda,IPIV,info1)
call dgetrs('n',n,NRHS,A,lda,IPIV,B,ldb,info2)
!$acc end host_data
!$acc end data
!
if ((info1.ne.0) .or. (info2.ne.0)) then
  print *,"test FAILED"
else if (any(abs(B(:,1)-1.0d0).gt.eps).or.any(abs(B(:,2)-2.0d0).gt.eps)) then
  print *,"test FAILED"
else
```

(continues on next page)

(continued from previous page)

```
  print *,"test PASSED"
end if
end
```

Here we show static arrays rather than dynamic arrays, so we use OpenACC data directives to control the data movement. The simplest compile and link line to use with this file is `nvfortran -acc=gpu -gpu=nvlamath testdgetrf.f90 -cudalib=nvlamath`.

13.6.4. NVLAMATH Usage From OpenMP

Usage of the NVLAmath functions from OpenMP are basically the same as from OpenACC. Use the `target data use_device_ptr` directive to pass device pointers to the functions.

```
program testdgetrf
integer, parameter :: M = 1000, N = M, NRHS=2
real(8) :: A(M,N), B(M,NRHS)
integer :: IPIV(M)
real(8), parameter :: eps = 1.0d-10
!
call random_number(A)
!$omp target enter data map(to:A) map(alloc:B,IPIV)
!$omp target teams loop
do i = 1, M
  A(i,i) = A(i,i) * 10.0d0
  B(i,1) = sum(A(i,:))
  B(i,2) = B(i,1) * 2.0d0
end do
!
lda = M; ldb = M
!$omp target data use_device_ptr(A, IPIV, B)
call dgetrf(M,N,A,lda,IPIV,info1)
call dgetrs('n',n,NRHS,A,lda,IPIV,B,ldb,info2)
!$omp end target data
!$omp target exit data map(from:B) map(delete:A,IPIV)
!
if ((info1.ne.0) .or. (info2.ne.0)) then
  print *,"test FAILED"
else if (any(abs(B(:,1)-1.0d0).gt.eps).or.any(abs(B(:,2)-2.0d0).gt.eps)) then
  print *,"test FAILED"
else
  print *,"test PASSED"
end if
end
```

Again we show static arrays rather than dynamic arrays, so we use OpenMP data directives to control the data movement. The simplest compile and link line to use with this file is `nvfortran -mp=gpu -gpu=nvlamath testdgetrf.f90 -cudalib=nvlamath`.

13.6.5. NVLAMATH List of Current Subroutines

Out of the thousands of LAPACK subroutines and functions, 31 of the most commonly used and requested subroutines are currently wrapped in this library.

Table 1: NVLAMATH List of Subroutines

		S	C	D	Z
Linear Solvers	LU	sgetrf	cgetrf	dgetrf	zgetrf
		sgetrs	cgetrs	dgetrs	zgetrs
		sgesv			
	Cholesky			dpotrf	zpotrf
				dpotrs	
Eigen Solvers	Eigen Solver	ssyev		dsyev	zheev
		ssyevd	cheevd	dsyevd	zheevd
	Partial Eigen Solver	ssyevx	cheevx	dsyevx	zheevx
		ssyevr		dsyevr	
	Eigen Solver for Generalized Problems			dsygv	zhegv
				dsygvd	zhegvd
	Partial Eigen Solver for Generalized Problems				zhegvx
	SVD			dgesvd	

13.6.6. NVLAMATH Argument Checks and CPU Fallback

In large, complicated applications, the compiler (and developers) can lose track of the attributes of data arrays: whether they are host arrays, managed, or device. Note that the compilers auto-conversion of allocatable arrays to managed require that they be allocatable. Consider the example from the drop-in case, but where the pivot array is declared statically.

```
program testdgetrf
integer, parameter :: M = 1000, N = M, NRHS=2
real(8), allocatable :: A(:,:), B(:,:)
integer :: IPIV(M)
real(8), parameter :: eps = 1.0d-10
!
allocate(A(M,N),B(M,NRHS))
call random_number(A)
do i = 1, M
    A(i,i) = A(i,i) * 10.0d0
    B(i,1) = sum(A(i,:))
    B(i,2) = B(i,1) * 2.0d0
end do
!
lda = M; ldb = M
call dgetrf(M,N,A,lda,IPIV,info1)
```

(continues on next page)

(continued from previous page)

```
call dgetrs('n',n,NRHS,A,lda,IPIV,B,ldb,info2)
!
if ((info1.ne.0) .or. (info2.ne.0)) then
    print *,"test FAILED"
else if (any(abs(B(:,1)-1.0d0) .gt. eps) .or. &
         any(abs(B(:,2)-2.0d0) .gt. eps)) then
    print *,"test FAILED"
else
    print *,"test PASSED"
end if
end
```

Now, we compile this as before, `nvfortran -stdpar -gpu=nvlamath testdgetrf.f90 -cudalib=nvlamath`. To review, `-stdpar` instructs the compiler and runtime to treat allocatable arrays as CUDA managed data. However, this time when the program is run:

```
** On entry to dgetrf parameter number 5 failed the pointer check
3: Accessible on GPU = T; Accessible on CPU = T
5: Accessible on GPU = F; Accessible on CPU = T
 Fallback to CPU compute disabled.
 Please (1) make sure that input arrays are accessible on the GPU; or (2) set
 environment variable NVCOMPILER_LAMATH_FALLBACK=1 to fallback to CPU execution.
 Terminate the application
```

Except when every array in the call is accessible only on the CPU, we assume that if the developer is using NVLAMATH, they intend to run the library function on the GPU, thus a mismatch in the GPU-accessibility of the subroutine arguments should be reported. As the error message states, you can continue through this error by setting the `NVCOMPILER_LAMATH_FALLBACK` environment variable to 1.

```
** On entry to dgetrf parameter number 5 failed the pointer check
3: Accessible on GPU = T; Accessible on CPU = T
5: Accessible on GPU = F; Accessible on CPU = T
 Fallback to CPU compute
** On entry to dgetrs parameter number 6 failed the pointer check
4: Accessible on GPU = T; Accessible on CPU = T
6: Accessible on GPU = F; Accessible on CPU = T
7: Accessible on GPU = T; Accessible on CPU = T
 Fallback to CPU compute
 test PASSED
```

Finally, we assume the developer would rather fix the issue by making the pivot array accessible on the GPU, either via CUDA Fortran device or managed attributes, OpenACC or OpenMP data directives, or changing it from statically declared to allocatable.

Chapter 14. Tensor Primitives Runtime Library APIs

This section describes the Fortran interfaces to the CUDA cuTENSOR library. The cuTENSOR functions are only accessible from host code. Most of the runtime API routines, other than some utilities, are functions that return an error code; they return a value of CUTENSOR_STATUS_SUCCESS if the call was successful, or another cuTENSOR status return value if there was an error. Unlike earlier Fortran modules, we have created a cutensorStatus derived type for the return values. We have also overloaded the `.eq.` and `.ne.` logical operators for testing the return status.

Currently we provide two levels of Fortran interfaces to the cuTENSOR library, a low-level module which maps 1-1 to the C interfaces in cuTENSOR v2.0.0, and an experimental high-level module which maps several standard Fortran intrinsic functions to the functionality contained within the cuTENSOR library.

The cuTENSOR documentation for version 2.0 explains how to migrate your low-level code from version 1.x to 2.x. Briefly, the create functions `cutensorCreateElementwiseBinary()`, `cutensorCreateElementwiseTrinary()`, `cutensorCreatePermutation()`, `cutensorCreateReduction()`, and `cutensorCreateContraction()` create an Operation Descriptor type. You create a plan preference and a plan using `cutensorCreatePlanPreference()` and `cutensorCreatePlan()`, respectively. Some operations require a workspace, which you can query using `cutensorEstimateWorkspaceSize()`. At this point, you have a cuTensor Plan type which can be passed into one of the execute functions `cutensorElementwiseBinaryExecute()`, `cutensorElementwiseTrinaryExecute()`, `cutensorPermute()`, `cutensorReduce()`, or `cutensorContract()`, along with the actual pointers to the device data, and scaling factors.

Chapter 12 contains examples of accessing the cuTENSOR library routines from OpenACC and CUDA Fortran. In both cases, the interfaces to the library can be exposed by adding the line

```
use cutensor_v2
```

to your program unit.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

14.1. CUTENSOR Definitions and Helper Functions

This section contains definitions and data types used in the cuTENSOR library and interfaces to the cuTENSOR helper functions.

Beginning with the NVHPC 24.1 release, the cuTENSOR version 2.x library is bundled in the release package. The version 2 API is significantly different than the previous 1.x releases, and the module name has been changed to avoid confusion. The new library interfaces can be exposed by adding the line

```
use cutensor_v2
```

to your program unit.

More than one cuTENSOR 2.x library is bundled, and the compiler driver selects one at link time based on the target architecture and the installed CUDA driver, so the version a program links against is not always the newest one present. The module declares the union of the APIs that those libraries provide. Where an API described below is not available in every bundled cuTENSOR version, its description states which version first supplied it. Calling `cutensorGetVersion()` reports the version actually loaded at run time, encoded as `major*10000 + minor*100 + patch`.

Using an entry point that the selected library predates is not always diagnosed. Most such entry points are simply absent from the older library and the link fails with an unresolved symbol. Users are advised to guard code that uses newer features, or require a minimum cuTENSOR version for the build.

This cuTENSOR module contains the following derived type definitions:

```
! Types from cutensor/types.h
! Algorithm Control
type, bind(c) :: cutensorAlgo
    integer(4) :: algo
end type
type(cutensorAlgo), parameter :: &
    CUTENSOR_ALGO_DEFAULT_PATIENT = cutensorAlgo(-6), &
    CUTENSOR_ALGO_GETT    = cutensorAlgo(-4), &
    CUTENSOR_ALGO_TGETT   = cutensorAlgo(-3), &
    CUTENSOR_ALGO_TTGT    = cutensorAlgo(-2), &
    CUTENSOR_ALGO_DEFAULT = cutensorAlgo(-1)
```

```
! Workspace Control
type, bind(c) :: cutensorWorksizePreference
    integer(4) :: wksp
end type
type(cutensorWorksizePreference), parameter :: &
    CUTENSOR_WORKSPACE_MIN         = cutensorWorksizePreference(1), &
    CUTENSOR_WORKSPACE_DEFAULT     = cutensorWorksizePreference(2), &
    CUTENSOR_WORKSPACE_MAX         = cutensorWorksizePreference(3)
```

```
! Unary and Binary Element-wise Operations
type, bind(c) :: cutensorOperator
    integer(4) :: opno
end type
type(cutensorOperator), parameter :: &
    ! Unary
```

(continues on next page)

(continued from previous page)

```
    CUTENSOR_OP_IDENTITY = cutensorOperator(1), & ! Identity operator
    CUTENSOR_OP_SQRT     = cutensorOperator(2), &  ! Square root
    CUTENSOR_OP_RELU     = cutensorOperator(8), &  ! Rectified linear unit
    CUTENSOR_OP_CONJ     = cutensorOperator(9), &  ! Complex conjugate
    CUTENSOR_OP_RCP      = cutensorOperator(10), & ! Reciprocal
    CUTENSOR_OP_SIGMOID = cutensorOperator(11), & ! y=1/(1+exp(-x))
    CUTENSOR_OP_TANH     = cutensorOperator(12), & ! y=tanh(x)
    CUTENSOR_OP_EXP      = cutensorOperator(22), & ! Exponentiation.
    CUTENSOR_OP_LOG      = cutensorOperator(23), & ! Log (base e).
    CUTENSOR_OP_ABS      = cutensorOperator(24), & ! Absolute value.
    CUTENSOR_OP_NEG      = cutensorOperator(25), & ! Negation.
    CUTENSOR_OP_SIN      = cutensorOperator(26), & ! Sine.
    CUTENSOR_OP_COS      = cutensorOperator(27), & ! Cosine.
    CUTENSOR_OP_TAN      = cutensorOperator(28), & ! Tangent.
    CUTENSOR_OP_SINH     = cutensorOperator(29), & ! Hyperbolic sine.
    CUTENSOR_OP_COSH     = cutensorOperator(30), & ! Hyperbolic cosine.
    CUTENSOR_OP_ASIN     = cutensorOperator(31), & ! Inverse sine.
    CUTENSOR_OP_ACOS     = cutensorOperator(32), & ! Inverse cosine.
    CUTENSOR_OP_ATAN     = cutensorOperator(33), & ! Inverse tangent.
    CUTENSOR_OP_ASINH    = cutensorOperator(34), & ! Inverse hyperbolic sine.
    CUTENSOR_OP_ACOSH    = cutensorOperator(35), & ! Inverse hyperbolic cosine.
    CUTENSOR_OP_ATANH    = cutensorOperator(36), & ! Inverse hyperbolic tangent.
    CUTENSOR_OP_CEIL     = cutensorOperator(37), & ! Ceiling.
    CUTENSOR_OP_FLOOR    = cutensorOperator(38), & ! Floor.
    CUTENSOR_OP_MISH     = cutensorOperator(39), & ! Mish y=x*tanh(softplus(x)).
    CUTENSOR_OP_SWISH    = cutensorOperator(40), & ! Swish y=x*sigmoid(x).
    CUTENSOR_OP_SOFT_PLUS = cutensorOperator(41), & ! Softplus y=log(exp(x)+1).
    CUTENSOR_OP_SOFT_SIGN = cutensorOperator(42), & ! Softsign y=x/(abs(x)+1).

    ! Binary
    CUTENSOR_OP_ADD      = cutensorOperator(3), &  ! Addition of two elements
    CUTENSOR_OP_MUL      = cutensorOperator(5), &  ! Multiplication of 2 elements
    CUTENSOR_OP_MAX      = cutensorOperator(6), &  ! Maximum of two elements
    CUTENSOR_OP_MIN      = cutensorOperator(7), &  ! Minimum of two elements
    CUTENSOR_OP_UNKNOWN = cutensorOperator(126)   ! reserved for internal use only
```

```
! Status Return Values
type, bind(c) :: cutensorStatus
    integer(4) :: stat
end type
type(cutensorStatus), parameter :: &
    ! The operation completed successfully.
    CUTENSOR_STATUS_SUCCESS                = cutensorStatus(0), &
    ! The cuTENSOR library was not initialized.
    CUTENSOR_STATUS_NOT_INITIALIZED        = cutensorStatus(1), &
    ! Resource allocation failed inside the cuTENSOR library.
    CUTENSOR_STATUS_ALLOC_FAILED           = cutensorStatus(3), &
    ! An unsupported value or parameter was passed to the function.
    CUTENSOR_STATUS_INVALID_VALUE          = cutensorStatus(7), &
    ! Indicates that the device is either not ready,
    ! or the target architecture is not supported.
    CUTENSOR_STATUS_ARCH_MISMATCH          = cutensorStatus(8), &
    ! An access to GPU memory space failed, which is usually caused
    ! by a failure to bind a texture.
    CUTENSOR_STATUS_MAPPING_ERROR          = cutensorStatus(11), &
    ! The GPU program failed to execute. This is often caused by a
```

(continues on next page)

(continued from previous page)

```
    ! launch failure of the kernel on the GPU, which can be caused by
    ! multiple reasons.
    CUTENSOR_STATUS_EXECUTION_FAILED       = cutensorStatus(13), &
    ! An internal cuTENSOR error has occurred.
    CUTENSOR_STATUS_INTERNAL_ERROR         = cutensorStatus(14), &
    ! The requested operation is not supported.
    CUTENSOR_STATUS_NOT_SUPPORTED          = cutensorStatus(15), &
    ! The functionality requested requires some license and an error
    ! was detected when trying to check the current licensing.
    CUTENSOR_STATUS_LICENSE_ERROR          = cutensorStatus(16), &
    ! A call to CUBLAS did not succeed.
    CUTENSOR_STATUS_CUBLAS_ERROR           = cutensorStatus(17), &
    ! Some unknown CUDA error has occurred.
    CUTENSOR_STATUS_CUDA_ERROR             = cutensorStatus(18), &
    ! The provided workspace was insufficient.
    CUTENSOR_STATUS_INSUFFICIENT_WORKSPACE = cutensorStatus(19), &
    ! Indicates that the driver version is insufficient.
    CUTENSOR_STATUS_INSUFFICIENT_DRIVER    = cutensorStatus(20), &
    ! Indicates an error related to file I/O
    CUTENSOR_STATUS_IO_ERROR               = cutensorStatus(21)
```

```
! Data Type
type, bind(c) :: cutensorDataType
    integer(4) :: cudaDataType
end type
type(cutensorDataType), parameter :: &
    CUTENSOR_R_16F  =  cutensorDataType(2), & ! real as a half
    CUTENSOR_C_16F  =  cutensorDataType(6), & ! complex as a pair of half numbers
    CUTENSOR_R_16BF = cutensorDataType(14), & ! real as a nv_bfloat16
    CUTENSOR_C_16BF = cutensorDataType(15), & ! complex as a pair of nv_bfloat16
↪numbers
    CUTENSOR_R_32F  =  cutensorDataType(0), & ! real as a float
    CUTENSOR_C_32F  =  cutensorDataType(4), & ! complex as a pair of float numbers
    CUTENSOR_R_64F  =  cutensorDataType(1), & ! real as a double
    CUTENSOR_C_64F  =  cutensorDataType(5), & ! complex as a pair of double numbers
    CUTENSOR_R_4I   = cutensorDataType(16), & ! real as a signed 4-bit int
    CUTENSOR_C_4I   = cutensorDataType(17), & ! complex as a pair of signed 4-bit int
↪numbers
    CUTENSOR_R_4U   = cutensorDataType(18), & ! real as a unsigned 4-bit int
    CUTENSOR_C_4U   = cutensorDataType(19), & ! complex as a pair of unsigned 4-bit
↪int numbers
    CUTENSOR_R_8I   =  cutensorDataType(3), & ! real as a signed 8-bit int
    CUTENSOR_C_8I   =  cutensorDataType(7), & ! complex as a pair of signed 8-bit int
↪numbers
    CUTENSOR_R_8U   =  cutensorDataType(8), & ! real as a unsigned 8-bit int
    CUTENSOR_C_8U   =  cutensorDataType(9), & ! complex as a pair of unsigned 8-bit
↪int numbers
    CUTENSOR_R_16I  = cutensorDataType(20), & ! real as a signed 16-bit int
    CUTENSOR_C_16I  = cutensorDataType(21), & ! complex as a pair of signed 16-bit int
↪numbers
    CUTENSOR_R_16U  = cutensorDataType(22), & ! real as a unsigned 16-bit int
    CUTENSOR_C_16U  = cutensorDataType(23), & ! complex as a pair of unsigned 16-bit
↪int numbers
    CUTENSOR_R_32I  = cutensorDataType(10), & ! real as a signed 32-bit int
    CUTENSOR_C_32I  = cutensorDataType(11), & ! complex as a pair of signed 32-bit int
↪numbers
```

(continues on next page)

(continued from previous page)

```
    CUTENSOR_R_32U  = cutensorDataType(12), & ! real as a unsigned 32-bit int
    CUTENSOR_C_32U  = cutensorDataType(13), & ! complex as a pair of unsigned 32-bit
↪int numbers
    CUTENSOR_R_64I  = cutensorDataType(24), & ! real as a signed 64-bit int
    CUTENSOR_C_64I  = cutensorDataType(25), & ! complex as a pair of signed 64-bit int
↪numbers
    CUTENSOR_R_64U  = cutensorDataType(26), & ! real as a unsigned 64-bit int
    CUTENSOR_C_64U  = cutensorDataType(27)        ! complex as a pair of unsigned 64-
↪bit int numbers
```

```
! Compute Type.  Superseded by cutensorComputeDescriptor and the
! CUTENSOR_COMPUTE_DESC_* values listed below; no version 2 entry point
! takes this type, but the module still declares it.
type, bind(c) :: cutensorComputeType
    integer(4) :: type
end type
type(cutensorComputeType), parameter :: &
    CUTENSOR_COMPUTE_16F    = cutensorComputeType(2**0), &  ! half
    CUTENSOR_COMPUTE_16BF   = cutensorComputeType(2**10), & ! bfloat16
    CUTENSOR_COMPUTE_TF32   = cutensorComputeType(2**12), & ! tf32
    CUTENSOR_COMPUTE_3XTF32 = cutensorComputeType(2**13), & ! 3xtf32
    CUTENSOR_COMPUTE_32F    = cutensorComputeType(2**2), &  ! float
    CUTENSOR_COMPUTE_64F    = cutensorComputeType(2**4), &  ! double
    CUTENSOR_COMPUTE_8U     = cutensorComputeType(2**6), &  ! uint8
    CUTENSOR_COMPUTE_8I     = cutensorComputeType(2**8), &  ! int8
    CUTENSOR_COMPUTE_32U    = cutensorComputeType(2**7), &  ! uint32
    CUTENSOR_COMPUTE_32I    = cutensorComputeType(2**9)     ! int32
```

```
! Operation Descriptor attribute
type, bind(c) :: cutensorOperationDescriptorAttribute
    integer(4) :: attr
end type
type(cutensorOperationDescriptorAttribute), parameter :: &
    CUTENSOR_OPERATION_DESCRIPTOR_TAG = &
      cutensorOperationDescriptorAttribute(0), &  ! integer(4)
    CUTENSOR_OPERATION_DESCRIPTOR_SCALAR_TYPE = &
      cutensorOperationDescriptorAttribute(1), &  ! cutensorDataType
    CUTENSOR_OPERATION_DESCRIPTOR_FLOPS = &
      cutensorOperationDescriptorAttribute(2), &  ! real(4)
    CUTENSOR_OPERATION_DESCRIPTOR_MOVED_BYTES = &
      cutensorOperationDescriptorAttribute(3), &  ! real(4)
    CUTENSOR_OPERATION_DESCRIPTOR_PADDING_LEFT = &
      cutensorOperationDescriptorAttribute(4), &  ! integer(4)
    CUTENSOR_OPERATION_DESCRIPTOR_PADDING_RIGHT = &
      cutensorOperationDescriptorAttribute(5), &  ! integer(4)
    CUTENSOR_OPERATION_DESCRIPTOR_PADDING_VALUE = &
      cutensorOperationDescriptorAttribute(6), &  ! type(c_ptr)
    ! New in cuTENSOR 2.7.0.  Set on a block-sparse contraction descriptor
    ! to request bitwise-reproducible results, at a performance cost.
    CUTENSOR_OPERATION_DESCRIPTOR_BLOCKSPARSE_REPRODUCIBLE = &
      cutensorOperationDescriptorAttribute(14)    ! integer(4)
```

```
! Plan Preference attribute
type, bind(c) :: cutensorPlanPreferenceAttribute
    integer(4) :: attr
end type
```

(continues on next page)

(continued from previous page)

```
type(cutensorPlanPreferenceAttribute), parameter :: &
    CUTENSOR_PLAN_PREFERENCE_AUTOTUNE_MODE = &
      cutensorPlanPreferenceAttribute(0), &  ! cutensorAutotuneMode type
    CUTENSOR_PLAN_PREFERENCE_CACHE_MODE = &
      cutensorPlanPreferenceAttribute(1), &  ! cutensorCacheMode type
    CUTENSOR_PLAN_PREFERENCE_INCREMENTAL_COUNT = &
      cutensorPlanPreferenceAttribute(2), &  ! integer(4)
    CUTENSOR_PLAN_PREFERENCE_ALGO = &
      cutensorPlanPreferenceAttribute(3), &  ! cutensorAlgo type
    CUTENSOR_PLAN_PREFERENCE_KERNEL_RANK = &
      cutensorPlanPreferenceAttribute(4), &  ! integer(4)
    CUTENSOR_PLAN_PREFERENCE_JIT = &
      cutensorPlanPreferenceAttribute(5), &  ! cutensorJitMode type
    ! New in cuTENSOR 2.6.0.  Plan for a specific GPU architecture rather
    ! than the one associated with the context.  The value encodes the SM
    ! version as 10 * SM.major + SM.minor.
    CUTENSOR_PLAN_PREFERENCE_GPU_ARCH = &
      cutensorPlanPreferenceAttribute(6)     ! integer(4)
```

```
! Plan attribute
type, bind(c) :: cutensorPlanAttribute
    integer(4) :: attr
end type
type(cutensorPlanAttribute), parameter :: &
    ! Pass integer(8), return exact required workspace in bytes needed
    CUTENSOR_PLAN_REQUIRED_WORKSPACE = cutensorPlanAttribute(0)
```

```
! Autotune Mode
type, bind(c) :: cutensorAutotuneMode
    integer(4) :: mode
end type
type(cutensorAutotuneMode), parameter :: &
    CUTENSOR_AUTOTUNE_MODE_NONE        = cutensorAutotuneMode(0), &
    CUTENSOR_AUTOTUNE_MODE_INCREMENTAL = cutensorAutotuneMode(1)
```

```
! JIT Mode
type, bind(c) :: cutensorJitMode
    integer(4) :: mode
end type
type(cutensorJitMode), parameter :: &
    CUTENSOR_JIT_MODE_NONE        = cutensorJitMode(0), &
    CUTENSOR_JIT_MODE_DEFAULT     = cutensorJitMode(1)
```

```
! Cache Mode
type, bind(c) :: cutensorCacheMode
    integer(4) :: mode
end type
type(cutensorCacheMode), parameter :: &
    CUTENSOR_CACHE_MODE_NONE        = cutensorCacheMode(0), &
    CUTENSOR_CACHE_MODE_PEDANTIC    = cutensorCacheMode(1)
```

```
! New 2.0 handle types
type, BIND(C) :: cutensorHandle
  TYPE(C_PTR) :: handle
end type
```

(continues on next page)

(continued from previous page)

```
type, BIND(C) :: cutensorTensorDescriptor
  TYPE(C_PTR) :: desc
end type

type, BIND(C) :: cutensorOperationDescriptor
  TYPE(C_PTR) :: desc
end type

type, BIND(C) :: cutensorComputeDescriptor
  TYPE(C_PTR) :: desc
end type

type, BIND(C) :: cutensorPlan
  TYPE(C_PTR) :: desc
end type

type, BIND(C) :: cutensorPlanPreference
  TYPE(C_PTR) :: desc
end type

! Block-sparse tensor descriptor, new in cuTENSOR 2.6.0
type, BIND(C) :: cutensorBlockSparseTensorDescriptor
  TYPE(C_PTR) :: desc
end type
```

```
! These are global C symbols in libcutensor
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_16F') ::  cutensor_Compute_Desc_16F
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_16BF') :: cutensor_Compute_Desc_16BF
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_TF32') :: cutensor_Compute_Desc_TF32
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_3XTF32') :: cutensor_Compute_Desc_3XTF32
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_32F') :: cutensor_Compute_Desc_32F
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_64F') :: cutensor_Compute_Desc_64F
! Emulated precisions, new in cuTENSOR 2.6.0
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_9X16BF') :: cutensor_Compute_Desc_9X16BF
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_8XINT8') :: cutensor_Compute_Desc_8XINT8
type(cutensorComputeDescriptor), &
  bind(C, name='CUTENSOR_COMPUTE_DESC_4X16F') :: cutensor_Compute_Desc_4X16F
```

The last three descriptors are new in cuTENSOR 2.6.0 and emulate a wider precision by composing several narrower values. Each is valid only with the precision it emulates: `CUTENSOR_COMPUTE_DESC_9X16BF` and `CUTENSOR_COMPUTE_DESC_4X16F` emulate single precision and are used with `CUTENSOR_R_32F` or `CUTENSOR_C_32F` tensors, while `CUTENSOR_COMPUTE_DESC_8XINT8` emulates double precision and is used with `CUTENSOR_R_64F` or `CUTENSOR_C_64F`. Pairing one with a precision it does not emulate, `CUTENSOR_COMPUTE_DESC_8XINT8` with single precision tensors for instance, is rejected when the operation descriptor is created.

Where the requested emulation is unavailable, either for the version of CUDA or for the target device, cuTENSOR falls back to native single or double precision compute. Results therefore may differ slightly

between devices.

14.1.1. cutensorCreate

This function initializes the cuTENSOR library and returns a handle for subsequent cuTENSOR calls.

```
type(cutensorStatus) function cutensorCreate(handle)
   type(cutensorHandle) :: handle
```

14.1.2. cutensorDestroy

This function releases the resources used by the cuTENSOR library. This function is usually the last call with a particular handle to the cuTENSOR API.

```
type(cutensorStatus) function cutensorDestroy(handle)
   type(cutensorHandle) :: handle
```

14.1.3. cutensorCreateTensorDescriptor

This function creates and initializes a cuTENSOR descriptor, given the number of modes, extents, strides, and type of the data.

```
type(cutensorStatus) function cutensorCreateTensorDescriptor(handle, desc, &
        numModes, extent, stride, dataType, alignmentRequirement)
    type(cutensorHandle) :: handle
    type(cutensorTensorDescriptor) :: desc
    integer(4) :: numModes
    integer(8), dimension(*) :: extent
    integer(8), dimension(*) :: stride
    type(cutensorDataType) :: dataType
    integer(4) :: alignmentRequirement
```

14.1.4. cutensorDestroyTensorDescriptor

This function deletes the cuTENSOR tensor descriptor and frees the resources associated with it.

```
type(cutensorStatus) function cutensorDestroyTensorDescriptor(desc)
    type(cutensorTensorDescriptor) :: desc
```

14.1.5. cutensorOperationDescriptorGetAttribute

This function retrieves an attribute of the Operation Descriptor type

```
type(cutensorStatus) function cutensorOperationDescriptorGetAttribute(handle, &
    desc, attr, buf, sizeInBytes)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorOperationDescriptorAttribute) :: attr
    integer(4) :: buf(*)  ! Any type, of size sizeInBytes
    integer(8) :: sizeInBytes
```

14.1.6. cutensorOperationDescriptorSetAttribute

This function sets an attribute of an Operation Descriptor type

```
type(cutensorStatus) function cutensorOperationDescriptorSetAttribute(handle, &
    desc, attr, buf, sizeInBytes)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorOperationDescriptorAttribute) :: attr
    integer(4) :: buf(*)  ! Any type, of size sizeInBytes
    integer(8) :: sizeInBytes
```

14.1.7. cutensorDestroyOperationDescriptor

This function frees the resources related to the Operation Descriptor type

```
type(cutensorStatus) function cutensorDestroyOperationDescriptor(desc)
    type(cutensorOperationDescriptor) :: desc
```

14.1.8. cutensorCreatePlanPreference

This function allocates and sets a cuTENSOR plan preference type.

```
type(cutensorStatus) function cutensorCreatePlanPreference(handle, pref, algo, &
        jitMode)
    type(cutensorHandle) :: handle
    type(cutensorPlanPreference) :: pref
    type(cutensorAlgo) :: algo
    type(cutensorJitMode) :: jitMode
```

14.1.9. cutensorDestroyPlanPreference

This function frees the resources associated with a plan preference type.

```
type(cutensorStatus) function cutensorDestroyPlanPreference(pref)
    type(cutensorPlanPreference) :: pref
```

14.1.10. cutensorPlanPreferenceSetAttribute

This function sets an attribute of an Operation Descriptor type

```
type(cutensorStatus) function cutensorPlanPreferenceSetAttribute(handle, pref, &
    attr, buf, sizeInBytes)
    type(cutensorHandle) :: handle
    type(cutensorPlanPreference) :: pref
    type(cutensorPlanPreferenceAttribute) :: attr
    integer(4) :: buf(*)  ! Any type, of size sizeInBytes
    integer(8) :: sizeInBytes
```

14.1.11. cutensorPlanPreferenceGetAttribute

This function retrieves an attribute of a Plan Preference type. The value is written into `buf`, which must be large enough to hold the attribute; pass its size in bytes as `sizeInBytes`. Use the same attribute constants listed for `cutensorPlanPreferenceSetAttribute()`.

```
! requires cuTENSOR 2.6.0 or newer
type(cutensorStatus) function cutensorPlanPreferenceGetAttribute(handle, pref, &
    attr, buf, sizeInBytes)
    type(cutensorHandle) :: handle
    type(cutensorPlanPreference) :: pref
    type(cutensorPlanPreferenceAttribute) :: attr
    integer(4) :: buf(*)  ! Any type, of size sizeInBytes
    integer(8) :: sizeInBytes
```

14.1.12. cutensorEstimateWorkspaceSize

This function determines the size of the required workspace for a given operation and preferences.

```
type(cutensorStatus) function cutensorEstimateWorkspaceSize(handle, desc, &
        pref, workspacePref, workspaceSizeEstimate)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorPlanPreference) :: pref
    type(cutensorWorksizePreference) :: workspacePref
    integer(8) :: workspaceSizeEstimate
```

14.1.13. cutensorCreatePlan

This function allocates and sets a cuTENSOR plan type.

```
type(cutensorStatus) function cutensorCreatePlan(handle, plan, &
        desc, pref, workspaceSizeLimit)
    type(cutensorHandle) :: handle
    type(cutensorPlan), intent(out) :: plan
    type(cutensorOperationDescriptor) :: desc
    type(cutensorPlanPreference) :: pref
    integer(8) :: workspaceSizeLimit
```

14.1.14. cutensorDestroyPlan

This function frees the resources associated with a cuTENSOR plan type.

```
type(cutensorStatus) function cutensorDestroyPlan(plan)
    type(cutensorPlan) :: plan
```

14.1.15. cutensorPlanGetAttribute

This function retrieves an attribute of a cuTENSOR plan type. The only attribute currently defined is `CUTENSOR_PLAN_REQUIRED_WORKSPACE`, which returns the exact workspace size in bytes that the plan requires. Pass an `integer(8)` as `buf` and `8` as `bufSizeInBytes`. Unlike `cutensorEstimateWorkspaceSize()`, which is called before the plan exists and returns an upper bound for a workspace preference, this reports what the created plan will actually use.

```
type(cutensorStatus) function cutensorPlanGetAttribute(handle, plan, &
    attr, buf, bufSizeInBytes)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    type(cutensorPlanAttribute) :: attr
    integer(4) :: buf(*)  ! Any type, of size bufSizeInBytes
    integer(8) :: bufSizeInBytes
```

14.1.16. cutensorGetErrorString

This function returns the description string for an error code.

```
character*128 function cutensorGetErrorString(ierr)
   type(cutensorStatus) :: ierr
```

14.1.17. cutensorGetVersion

This function returns the version number of the cuTENSOR library.

```
integer(8) function cutensorGetVersion()
```

14.1.18. cutensorGetCudartVersion

This function returns the version of the CUDA runtime that the cuTENSOR library was compiled against.

```
integer(8) function cutensorGetCudartVersion()
```

14.2. CUTENSOR Plan and Kernel Cache Operations

This section contains interfaces to the two caches cuTENSOR maintains per handle. They are independent of each other.

The *plan cache* is a software cache of created plans, held in host memory. It lets a repeated operation reuse an existing plan instead of running the planning phase again. Every handle is created with a plan cache already enabled, holding the 64 least recently used plans; use `cutensorHandleResizePlanCache()` to change that capacity. The cache can be written to a file and read back, so that a later run of the same program starts with it populated.

The *kernel cache* holds kernels produced by just-in-time compilation. It is only useful when JIT is enabled through the `CUTENSOR_PLAN_PREFERENCE_JIT` plan preference attribute, and the kernel cache entry points return an error status on a device or toolkit that does not support JIT compilation.

The four file-based entry points take an ordinary Fortran `character` argument.

14.2.1. cutensorHandleResizePlanCache

This function sets the number of entries in the plan cache attached to a handle. A handle is created with a cache of 64 entries, so this is only needed to choose a different capacity; passing zero disables the cache. Resizing invalidates the cache, discarding any plans it held.

```
type(cutensorStatus) function cutensorHandleResizePlanCache(handle, numEntries)
    type(cutensorHandle) :: handle
    integer(4) :: numEntries
```

14.2.2. cutensorHandleWritePlanCacheToFile

This function writes the current contents of the plan cache to the named file.

```
type(cutensorStatus) function cutensorHandleWritePlanCacheToFile(handle, filename)
    type(cutensorHandle) :: handle
    character*(*) :: filename
```

14.2.3. cutensorHandleReadPlanCacheFromFile

This function reads plan cache entries from the named file into the handle's plan cache, and returns the number of cache lines it read in `numCachelinesRead`. If the cache is too small to hold the file's contents, the function returns the number of lines the cache would need, so a caller can resize and read again.

```
type(cutensorStatus) function cutensorHandleReadPlanCacheFromFile(handle, &
    filename, numCachelinesRead)
    type(cutensorHandle) :: handle
    character*(*) :: filename
    integer(4), intent(out) :: numCachelinesRead
```

14.2.4. cutensorWriteKernelCacheToFile

This function writes the just-in-time compiled kernels held by the handle to the named file.

```
type(cutensorStatus) function cutensorWriteKernelCacheToFile(handle, filename)
    type(cutensorHandle) :: handle
    character*(*) :: filename
```

14.2.5. cutensorReadKernelCacheFromFile

This function reads previously compiled kernels from the named file, so that a plan created with JIT enabled can reuse them instead of compiling again.

```
type(cutensorStatus) function cutensorReadKernelCacheFromFile(handle, filename)
    type(cutensorHandle) :: handle
    character*(*) :: filename
```

14.3. CUTENSOR Logging

This section contains interfaces to the cuTENSOR logger. Logging can also be turned on without changing your program, by setting the `CUTENSOR_LOG_LEVEL` environment variable before the run.

Unlike the rest of this module, these functions return a plain `integer(4)` rather than a `type(cutensorStatus)`. A return of zero means success.

Two of them are extensions to the C API provided for Fortran convenience: `cutensorLoggerOpenFile()` takes an ordinary Fortran `character` file name, and `cutensorLoggerSetFile()` takes a Fortran unit number that your program has already opened.

14.3.1. cutensorLoggerSetLevel

This function sets how much the logger reports. The level should be one of 0 for off, 1 for errors, 2 for performance traces, 3 for performance hints, 4 for heuristics traces, or 5 for API traces. Values outside 0 to 5 are rejected.

```
integer(4) function cutensorLoggerSetLevel(level)
    integer :: level
```

14.3.2. cutensorLoggerSetMask

This function selects which classes of message are logged, independently of the level. The mask is the sum of 1 for errors, 2 for performance traces, 4 for performance hints, 8 for heuristics traces and 16 for API traces.

```
integer(4) function cutensorLoggerSetMask(mask)
    integer :: mask
```

14.3.3. cutensorLoggerOpenFile

This function redirects the log to a file, which the library opens by name.

```
integer(4) function cutensorLoggerOpenFile(logFile)
    character*(*) :: logFile
```

14.3.4. cutensorLoggerSetFile

This function redirects the log to a Fortran unit that your program has already opened.

The library takes ownership of the underlying file for as long as it is logging to it. Do not close the unit while the logger still points at it; close it only after `cutensorLoggerForceDisable()`, or after redirecting the log elsewhere.

```
integer(4) function cutensorLoggerSetFile(unit)
    integer :: unit
```

14.3.5. cutensorLoggerSetCallback

This function installs a procedure that the library calls for each log message, instead of writing it to a file. Pass the address of the procedure using `c_funloc()` from the `iso_c_binding` module.

The callback must be `bind(C)` and take three arguments by value: the log level as an `integer(c_int32_t)`, and the function name and message as `type(c_ptr)`. Use `c_f_pointer()` to read the two strings, which are null-terminated.

If a variable that the callback updates is examined by the calling program, declare that variable `volatile`.

```
integer(4) function cutensorLoggerSetCallback(callback)
    type(c_funptr) :: callback
```

14.3.6. cutensorLoggerForceDisable

This function turns logging off for the remainder of the run. It cannot be undone, so call it last.

```
integer(4) function cutensorLoggerForceDisable()
```

14.4. CUTENSOR Element-wise Operations

This section contains interfaces for the cuTENSOR functions that perform element-wise operations between tensors.

14.4.1. cutensorCreatePermutation

This function creates an out-of-place tensor permutation operator of the form

`B = alpha * opA(perm(A))` The permutation operation information is stored in the intent(out) `desc` argument.

The arrays `A, B` can be of any supported type, kind, and rank. The permutations of `A, B` are set up using the mode arguments.

```
type(cutensorStatus) function cutensorCreatePermutation(handle, desc, &
    descA, modeA, opA, descB, modeB, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor), intent(out) :: desc
    type(cutensorTensorDescriptor) :: descA, descB
    integer(4), dimension(*) :: modeA, modeB
    type(cutensorOperator) :: opA
    type(cutensorComputeDescriptor) :: descCompute
```

14.4.2. cutensorPermute

This function executes an out-of-place tensor permutation of the form

`B = alpha * opA(perm(A))`

The type and kind of the `alpha` scalar is determined by the typeScalar argument. The arrays `A, B` can be of any supported type, kind, and rank. The permutations of `A, B` are set up using the mode arguments. The operation opA(perm(A)) is set up in the call to `cutensorInitTensorDescriptor()` via the `unaryOp` argument.

```
type(cutensorStatus) function cutensorPermute(handle, plan, &
    alpha, A, B, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha
    real, device, dimension(*) :: A, B
    integer(kind=cuda_stream_kind) :: stream
```

14.4.3. cutensorCreateElementwiseBinary

This function creates an element-wise tensor operation on two inputs of the form

`D=opAC(alpha*op(perm(A)),gamma*op(perm(C)))`

The `opA, opC` arguments are an element-wise unary operator. The `opAC` argument is an element-wise binary operator. The arrays `A, C, D` can be of any supported type, kind, and rank. The permutations of `A, C, D` are set up using the mode arguments.

```
type(cutensorStatus) function cutensorCreateElementwiseBinary(handle, desc, &
        descA, modeA, opA, descC, modeC, opC, &
        descD, modeD, opAC, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorTensorDescriptor) :: descA, descC, descD
    integer(4), dimension(*) :: modeA, modeC, modeD
    type(cutensorOperator) :: opA, opC, opAC
    type(cutensorComputeDescriptor) :: descCompute
```

14.4.4. cutensorElementwiseBinaryExecute

This function executes an element-wise tensor operation on two inputs of the form

`D=opAC(alpha*op(perm(A)),gamma*op(perm(C)))`

The specified plan is created by a call to `cutensorCreatePlan()`. The arrays `A, C, D` can be of any supported type, kind, and rank. The allowable type and kind of the `alpha, gamma` scalars is determined by the compute descriptor in the call to `cutensorCreateElementwiseBinary`.

```
type(cutensorStatus) function cutensorElementwiseBinaryExecute(handle, &
    plan, alpha, A, gamma, C, D, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, gamma  ! Any compatible type and kind
    real, device, dimension(*) :: A, C, D
    integer(kind=cuda_stream_kind) :: stream
```

14.4.5. cutensorCreateElementwiseTrinary

This function creates an element-wise tensor operation on three inputs of the form

`D=opABC(opAB(alpha*op(perm(A)),beta*op(perm(B))),gamma*op(perm(C)))`

The `opA, opB, opC` arguments are an element-wise unary operator. The `opAB` argument is an element-wise binary operator. The arrays `A, B, C, D` can be of any supported type, kind, and rank. The permutations of `A, B, C, D` are set up using the mode arguments.

```
type(cutensorStatus) function cutensorCreateElementwiseTrinary(handle, desc, &
        descA, modeA, opA, descB, modeB, opB, descC, modeC, opC, &
        descD, modeD, opAB, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorTensorDescriptor) :: descA, descB, descC, descD
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD
    type(cutensorOperator) :: opA, opB, opC, opAB
    type(cutensorComputeDescriptor) :: descCompute
```

14.4.6. cutensorElementwiseTrinaryExecute

This function executes an element-wise tensor operation on three inputs of the form

`D=opABC(opAB(alpha*op(perm(A)),beta*op(perm(B))),gamma*op(perm(C)))`

The specified plan is created by a call to `cutensorCreatePlan()`. The arrays `A, B, C, D` can be of any supported type, kind, and rank. The allowable type and kind of the `alpha, gamma` scalars is determined by the compute descriptor in the call to `cutensorCreateElementwiseTrinary`.

```
type(cutensorStatus) function cutensorElementwiseTrinaryExecute(handle, plan,  &
    alpha, A, beta, B, gamma, C, D, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, beta, gamma  ! Any compatible type and kind
```

(continues on next page)

(continued from previous page)

```
    real, device, dimension(*) :: A, B, C, D
    integer(kind=cuda_stream_kind) :: stream
```

14.5. CUTENSOR Reduction Operations

This section contains interfaces for the cuTENSOR functions that perform reduction operations on tensors.

14.5.1. cutensorCreateReduction

This function creates a reduction tensor operation of the form:

`D = alpha * opReduce(opA(A)) + beta * opC(C)`

The `opA, opC` arguments are an element-wise unary operator. The `opReduce` argument is a binary operator. The arrays `A, C, D` can be of any supported type, kind, and rank. The permutations of `A, C` are set up using the mode arguments.

```
type(cutensorStatus) function cutensorCreateReduction(handle, desc, &
        descA, modeA, opA, descC, modeC, opC, &
        descD, modeD, opReduce, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorTensorDescriptor) :: descA, descC, descD
    integer(4), dimension(*) :: modeA, modeC, modeD
    type(cutensorOperator) :: opA, opC, opReduce
    type(cutensorComputeDescriptor) :: descCompute
```

14.5.2. cutensorReduce

This function executes a reduction of the form:

`D = alpha * opReduce(opA(A)) + beta * opC(C)`

The specified plan is created by a call to `cutensorCreatePlan()`. The arrays `A, C, D` can be of any supported type, kind, and rank. The allowable type and kind of the `alpha, beta` scalars is determined by the compute descriptor in the call to `cutensorCreateReduction`.

```
type(cutensorStatus) function cutensorReduce(handle, plan, &
    alpha, A, beta, C, D, workspace, workspaceSize, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, beta  ! Any compatible type and kind
    real, device, dimension(*) :: A, C, D
    real, device, dimension(*) :: workspace  ! Any type
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

14.6. CUTENSOR Contraction Operations

This section contains interfaces for the cuTENSOR functions that perform contraction operations between tensors.

14.6.1. cutensorCreateContraction

This function creates a contraction tensor operation of the form:

`D = alpha*(AxB) + beta*C`

The `opA, opB, opC` arguments are an element-wise unary operator. The arrays `A, B, C, D` can be of any supported type, kind, and rank. The permutations of `A, B, C` are set up using the mode arguments.

```
type(cutensorStatus) function cutensorCreateContraction(handle, desc, &
        descA, modeA, opA, descB, modeB, opB, &
        descC, modeC, opC, descD, modeD, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor) :: desc
    type(cutensorTensorDescriptor) :: descA, descB, descC, descD
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD
    type(cutensorOperator) :: opA, opB, opC
    type(cutensorComputeDescriptor) :: descCompute
```

14.6.2. cutensorContract

This function executes a contraction of the form:

`D = alpha*(AxB) + beta*C`

The specified plan is created by a call to `cutensorCreatePlan()`. The arrays `A, B, C, D` can be of any supported type, kind, and rank. The allowable type and kind of the `alpha, beta` scalars is determined by the compute descriptor in the call to `cutensorCreateContraction`.

```
type(cutensorStatus) function cutensorContract(handle, plan, &
    alpha, A, B, beta, C, D, workspace, workspaceSize, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, beta  ! Any compatible type and kind
    real, device, dimension(*) :: A, B, C, D
    real, device, dimension(*) :: workspace  ! Any type
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

14.6.3. cutensorCreateContractionTrinary

This function creates an Operation Descriptor for a contraction of three input tensors, of the form:

`E = alpha*(AxBxC) + beta*D`

The three inputs are contracted in one operation rather than as two successive binary contractions, which avoids materializing the intermediate result. Each of the five tensors has its own descriptor and mode array, and each of the four inputs has its own element-wise operator.

```
type(cutensorStatus) function cutensorCreateContractionTrinary(handle, desc, &
    descA, modeA, opA, descB, modeB, opB, descC, modeC, opC, &
    descD, modeD, opD, descE, modeE, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor), intent(out) :: desc
    type(cutensorTensorDescriptor) :: descA, descB, descC, descD, descE
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD, modeE
    type(cutensorOperator) :: opA, opB, opC, opD
    type(cutensorComputeDescriptor) :: descCompute
```

14.6.4. cutensorContractTrinary

This function executes a contraction of three input tensors, of the form:

`E = alpha*(AxBxC) + beta*D`

The specified plan is created by a call to `cutensorCreatePlan()` on a descriptor from `cutensorCreateContractionTrinary()`. The arrays `A, B, C, D, E` can be of any supported type, kind, and rank. The allowable type and kind of the `alpha, beta` scalars is determined by the compute descriptor used to create the descriptor.

```
type(cutensorStatus) function cutensorContractTrinary(handle, plan, &
    alpha, A, B, C, beta, D, E, workspace, workspaceSize, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, beta  ! Any compatible type and kind
    real, device, dimension(*) :: A, B, C, D, E
    real, device, dimension(*) :: workspace  ! Any type
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

14.6.5. cutensorCreateBlockSparseTensorDescriptor

This function creates a descriptor for a block-sparse tensor, one whose modes are divided into sections so that the tensor is a grid of blocks, only some of which are stored.

Each mode is split into `numSectionsPerMode` sections, so `numSectionsPerMode` has one element per mode. `extent` holds the extent of every section, the sections of the first mode first, then those of the second, and so on, so its length is the sum of `numSectionsPerMode`. `nonZeroCoordinates` gives the block coordinates of each stored block, `numModes` values per block, and the blocks may be listed in any order. `stride` gives the strides of each stored block in the same block order, again `numModes` values per block. A maximum of 32 modes is supported, and `dataType` must be one of `CUTENSOR_R_32F`, `CUTENSOR_R_64F`, `CUTENSOR_C_32F` or `CUTENSOR_C_64F`.

```
! requires cuTENSOR 2.6.0 or newer
type(cutensorStatus) function cutensorCreateBlockSparseTensorDescriptor(handle, &
    desc, numModes, numNonZeroBlocks, numSectionsPerMode, extent, &
    nonZeroCoordinates, stride, dataType)
    type(cutensorHandle) :: handle
    type(cutensorBlockSparseTensorDescriptor), intent(out) :: desc
    integer(4) :: numModes
    integer(8) :: numNonZeroBlocks
    integer(4), dimension(*) :: numSectionsPerMode, nonZeroCoordinates
    integer(8), dimension(*) :: extent, stride
    type(cutensorDataType) :: dataType
```

14.6.6. cutensorDestroyBlockSparseTensorDescriptor

This function frees the resources associated with a block-sparse tensor descriptor.

```
! requires cuTENSOR 2.6.0 or newer
type(cutensorStatus) function cutensorDestroyBlockSparseTensorDescriptor(desc)
    type(cutensorBlockSparseTensorDescriptor) :: desc
```

14.6.7. cutensorCreateBlockSparseContraction

This function creates an Operation Descriptor for a contraction of block-sparse tensors, of the form:

`D = alpha*(AxB) + beta*C`

Only the blocks declared non-zero in the descriptor of `D` are computed. Unlike the dense `cutensorCreateContraction()`, there is no operator argument for `D`.

Setting the `CUTENSOR_OPERATION_DESCRIPTOR_BLOCKSPARSE_REPRODUCIBLE` attribute on the resulting descriptor requests bitwise-reproducible results, at a performance cost. That attribute needs cuTENSOR 2.7.0, one release later than the block-sparse routines themselves.

```
! requires cuTENSOR 2.6.0 or newer
type(cutensorStatus) function cutensorCreateBlockSparseContraction(handle, desc, &
    descA, modeA, opA, descB, modeB, opB, descC, modeC, opC, &
    descD, modeD, descCompute)
    type(cutensorHandle) :: handle
    type(cutensorOperationDescriptor), intent(out) :: desc
    type(cutensorBlockSparseTensorDescriptor) :: descA, descB, descC, descD
    integer(4), dimension(*) :: modeA, modeB, modeC, modeD
    type(cutensorOperator) :: opA, opB, opC
    type(cutensorComputeDescriptor) :: descCompute
```

14.6.8. cutensorBlockSparseContract

This function executes a block-sparse contraction of the form:

`D = alpha*(AxB) + beta*C`

The blocks of each tensor are stored separately, so `A, B, C, D` are host arrays of device pointers with one entry per non-zero block, in the same block order used when the descriptor was created. Build each entry with `c_devloc()` applied to a device array.

Block-sparse execution also requires a recent GPU architecture. On a device that does not support it, the descriptors and the plan are still created successfully and this function returns `CUTENSOR_STATUS_NOT_SUPPORTED`.

```
! requires cuTENSOR 2.6.0 or newer
type(cutensorStatus) function cutensorBlockSparseContract(handle, plan, &
    alpha, A, B, beta, C, D, workspace, workspaceSize, stream)
    type(cutensorHandle) :: handle
    type(cutensorPlan) :: plan
    real :: alpha, beta  ! Any compatible type and kind
    type(c_devptr), dimension(*) :: A, B, C, D
    real, device, dimension(*) :: workspace  ! Any type
    integer(8) :: workspaceSize
    integer(kind=cuda_stream_kind) :: stream
```

14.7. CUTENSOR Fortran Extensions

This section contains extensions to the cuTENSOR interfaces for Fortran array intrinsic function operations, array expressions, and array assignment, built upon the cuTENSOR library. In CUDA Fortran, these operations take data with the device or managed attribute. In OpenACC, they can be invoked with a host_data construct. The interfaces to these extensions are exposed by adding the line `use cutensorEx` to your program unit, or can be applied to specific statements using the Fortran block feature, as shown in the next example.

```
block; use cutensorEx
  D = reshape(A,shape=[ni,nk,nj],order=[1,3,2])
end block
```

To enable the operations to take place in one underlying kernel invocation, the RHS expression is deferred until the overloaded assignment operation has both the LHS and RHS available. This guarantees performance on par with the low-level cuTENSOR API whenever possible. Generality is affected, so only specific forms of the Fortran statements are currently supported, and those are documented in the subsequent sections of this chapter.

Since the cuTENSOR library operations take a stream argument, we have added a way to set a cuTENSOR default stream that our runtime will maintain, one copy per CPU thread. That is:

```
integer function cutensorexSetStream(stream)
integer(kind=cuda_stream_kind) :: stream
```

14.7.1. Fortran Reshape

This Fortran function changes the shape of an array and possibly permutes the dimensions and layout. It is invoked as:

```
D = alpha * func(reshape(A, shape=[...], order=[...]))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A, or as func(reshape(A)), if that differs. Accepted functions which can be applied to the result of reshape are listed at the end of this section. The pad argument to the F90 reshape function is not currently supported. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
! Example to switch the 2nd and 3rd dimension layout
D = reshape(a,shape=[ni,nk,nj], order=[1,3,2])
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(reshape(a,shape=[ni,nk,nj], order=[1,3,2]))
```

14.7.2. Fortran Transpose

This Fortran function tranposes a matrix (a 2-dimensional array). It is invoked as:

```
D = alpha * func(transpose(A))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D is 2. Applying scaling (the `alpha` argument) or applying a function to the transpose result is optional. The alpha value is expected to be the same type as A, or as func(transpose(A)), if that differs. Accepted functions which can be applied to the result of the transpose are listed at the end of this section. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
! Example of transpose
D = transpose(A)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(tranpose(A))
```

14.7.3. Fortran Spread

This Fortran function increases the rank of an array by one across the specified dimension and broadcasts the values over the new dimension. It is invoked as:

```
D = alpha * func(spread(A, dim=i, ncopies=n))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A. Accepted functions which can be applied to the result of spread are listed at the end of this section. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
! Example to add and broadcast values over the new first dimension
D = spread(A, dim=1, ncopies=n1)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(spread(A, dim=1, ncopies=n1))
```

14.7.4. Fortran Element-wise Expressions

There is some limited support for converting expressions involving two or three source arrays into cuTENSOR calls. The first one or two operands can be a permuted array, the result of a call to reshape(), transpose(), or spread(). An elemental function can be applied to the array operands, permuted or not, and they can also be scaled. Here are some supported forms:

```
D = A + B
D = permute(A) + B
D = A + permute(B)
D = permute(A) - B
D = A - permute(B)
D = A + func(permute(B))
D = func(permute(A)) + permute(B)
D = alpha * func(permute(A)) + beta * permute(B) + gamma * C
```

The arrays A, B, C, and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A, B, C, and D can be from 1 to 7. For the three-operand case, arrays C and D must have the same shape, strides, and type. The alpha value is expected to be the same type as A. The same applies for beta and B, and gamma and C. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Accepted functions which can be applied to permuted or unpermuted arrays are listed at the end of this section. These Fortran expressions, besides initialization and setting up cuTENSOR descriptors, map to either `cutensorElementwiseBinary()` or `cutensorElementwiseTrinary()`.

```
! Example to scale and add two arrays together
D = alpha * A + beta * B
! Same example, take the absolute value of A and B and add to C
D = alpha * abs(A) + beta * abs(B) + C
! Transpose the first array before adding to the second
D = alpha * abs(transpose(A)) + beta * abs(B) + C
```

14.7.5. Fortran Matmul Operations

Matrix multiplication is one instance of tensor contraction. Either operand to matmul can be a permuted array, the result of a call to reshape(), transpose(), or spread(). The cuTENSOR library does not currently support applying an elemental function to the array operands, but the result and accumulator can be scaled. Here are some supported forms:

```
D = matmul(A, B)
D = matmul(permute(A), B)
```

```
D = matmul(A, permute(B))
D = matmul(permute(A), permute(B))
D = C + matmul(A, B)
D = C - matmul(A, B)
D = alpha * matmul(A, B) + beta * C
```

The arrays A, B, C, and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A, B, C, and D must be 2, after any permutations. Arrays C and D must currently have the same shape, strides, and type. The alpha value is expected to be the same type as A and B. The beta value should have the same type as C. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Fortran support for `Matmul`, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorContraction()`.

```
! Example to multiply two matrices together
D = matmul(A, B)
! Same example, accumulate into C
C = C +  matmul(A, B)
! Same example, transpose the first argument
C = C + matmul(transpose(A), B)
```

On GPUs which support the TF32 type, to direct a contraction to use TF32 rather than full single precision for real(4) (or similar for complex(4)), we have provided a way to set an internal parameter, similar to the default stream, that our runtime will maintain. The default opt level is 0. Setting the opt level to be greater than 0 will use TF32.

```
integer function cutensorExSetOptLevel(level)
integer(4) :: level
```

14.7.6. Fortran Dot_Product Operations

A Dot Product is another instance of tensor contraction. In this implementation, we have added a new `dim` argument to make dot_product more generally applicable to higher-order arrays. It follows very closely to the matmul features in the previous section. Either of the two operands to dot_product can be a permuted array, the result of a call to reshape(), transpose(), or spread(). Note that only reshape() can generate a 1-D array. The other calls can be used along with the dim argument. The cuTENSOR library does not currently support applying an elemental function to the array operands, but the result and accumlator can be scaled. Here are some supported forms:

```
X = dot_product(A, B)
X = dot_product(reshape(A,shape=[n]), B)
X = dot_product((A, reshape(B,shape=[n]))
D = dot_product(permute(A), permute(B),dim=i)
D = C + dot_product(A, B, dim=i)
D = C - dot_product(A, B, dim=i)
D = C + alpha * dot_product(A, B, dim=i)
```

The arrays A, B, C, and D and scalar X can be of type real(2), real(4), real(8), complex(4), or complex(8). Arrays C and D must currently have the same shape, strides, and type. The rank of D and C is one less than A and B for the case using the dim argument. The alpha value is expected to be the same type as A and B. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Fortran support for `Dot_Product`, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorContraction()`.

14.7.7. Supported Element-wise Functions

From the list of supported element-wise functions for the cuTENSOR definitions listed above, several are supported from the high-level interface. The cuTENSOR library does not currently support many of the functions listed for complex data; consult the cuTENSOR documentation for the latest information. Here are the functions with at least some level of Fortran interface support:

`SQRT RELU CONJG RCP SIGMOID`

`TANH EXP LOG ABS NEG`

`SIN COS TAN SINH COSH`

`ASIN ACOS ATAN ASINH ACOSH`

`ATANH CEIL FLOOR`

Note the C complex conjugate function `conj` is spelled `conjg` in Fortran. Also, the functions for `ceil` and `floor` differ between C and Fortran, in their return type. We have kept the C spelling and behavior (returning a real, not an integer).

Only `ABS, CEIL, CONJG, COS, SIN` can be used on bare arrays in the current implementation. All functions can be applied to the result of a permutation. Use `reshape(A, shape=shape(A))` as a NOP to put the bare array into a form that can currently be recognized.

Users will find that the performance of binary or trinary kernels, such as:

```
D = sin(A) + cos(B) + C
```

will not perform as well as kernels written and compiled for that specific operation (using CUDA, CUDA Fortran, or OpenACC) due to overhead in the cuTENSOR kernels needed for generally applying functions to the operands. If the operation permutes the operands, such as:

```
D = sin(permute(A)) + cos(permute(B)) + C
```

users may see good performance compared to other naive implementations depending on how complex the permutations on the input arrays are.

Chapter 15. NVIDIA Collective Communications Library (NCCL) APIs

This section describes the Fortran interfaces to the NCCL library. The NCCL functions are only accessible from host code. Most of the runtime API routines, other than some utilities, are functions that return an error code; they return a value of ncclSuccess if the call was successful, or another value if there was an error. Unlike earlier Fortran modules, we have created a ncclResult derived type for the return values. We have also overloaded the `.eq.` and `.ne.` logical operators for testing the return status.

The NCCL interfaces and definitions described in this chapter can be exposed in host code by adding the line

```
use nccl
```

to your program unit.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

15.1. NCCL Definitions and Helper Functions

This section contains definitions and data types used in the NCCL library and interfaces to the NCCL communicator creation and management functions.

The Fortran NCCL module contains the following derived type definitions:

```
! Definitions from nccl.h
integer, parameter :: NCCL_MAJOR = 2
integer, parameter :: NCCL_MINOR = 31
integer, parameter :: NCCL_PATCH = 2
integer, parameter :: NCCL_VERSION = &
    (NCCL_MAJOR * 10000 + NCCL_MINOR * 100 + NCCL_PATCH)
integer, parameter :: NCCL_VERSION_CODE = NCCL_VERSION

integer, parameter :: NCCL_SPLIT_NOCOLOR = -1

! Window registration flags
integer, parameter :: NCCL_WIN_DEFAULT = 0
```

(continues on next page)

(continued from previous page)

```
integer, parameter :: NCCL_WIN_COLL_SYMMETRIC = 1
integer, parameter :: NCCL_WIN_STRICT_ORDERING = 2
integer, parameter :: NCCL_WIN_REQUIRED_ALIGNMENT = 4096

! CTA scheduling policies
integer, parameter :: NCCL_CTA_POLICY_DEFAULT = 0
integer, parameter :: NCCL_CTA_POLICY_EFFICIENCY = 1
integer, parameter :: NCCL_CTA_POLICY_ZERO = 2

! Shrink, revoke and suspend flags
integer, parameter :: NCCL_SHRINK_DEFAULT = 0
integer, parameter :: NCCL_SHRINK_ABORT = 1
integer, parameter :: NCCL_REVOKE_DEFAULT = 0
integer, parameter :: NCCL_SUSPEND_MEM = 1

! Sentinels for configuration fields that are left unset
integer(c_int), parameter :: NCCL_CONFIG_UNDEF_INT = -2147483647 - 1
real(c_float), parameter :: NCCL_UNDEF_FLOAT = -1.0
type(c_ptr), parameter :: NCCL_CONFIG_UNDEF_PTR = c_null_ptr
integer(c_int), parameter :: NCCL_API_MAGIC = -889274641 ! z'cafebeef'
```

```
! Types from nccl.h
! ncclUniqueId
type, bind(c) :: ncclUniqueId
  character(c_char) :: internal(NCCL_UNIQUE_ID_BYTES)
end type ncclUniqueId

! ncclComm
type, bind(c) :: ncclComm
  type(c_ptr) :: member
end type ncclComm

type(ncclComm), parameter :: NCCL_COMM_NULL = ncclComm(c_null_ptr)

! ncclResult
type, bind(c) :: ncclResult
  integer(c_int) :: member
end type ncclResult

type(ncclResult), parameter :: &
    ncclSuccess             = ncclResult(0), &
    ncclUnhandledCudaError  = ncclResult(1), &
    ncclSystemError         = ncclResult(2), &
    ncclInternalError       = ncclResult(3), &
    ncclInvalidArgument     = ncclResult(4), &
    ncclInvalidUsage        = ncclResult(5), &
    ncclRemoteError         = ncclResult(6), &
    ncclInProgress          = ncclResult(7), &
    ncclTimeout             = ncclResult(8), &
    ncclNumResults          = ncclResult(9)
```

```
! ncclDataType
type, bind(c) :: ncclDataType
  integer(c_int) :: member
end type ncclDataType
```

(continues on next page)

(continued from previous page)

```
type(ncclDataType), parameter :: &
    ncclInt8       = ncclDataType(0), &
    ncclChar       = ncclDataType(0), &
    ncclUint8      = ncclDataType(1), &
    ncclInt32      = ncclDataType(2), &
    ncclInt        = ncclDataType(2), &
    ncclUint32     = ncclDataType(3), &
    ncclInt64      = ncclDataType(4), &
    ncclUint64     = ncclDataType(5), &
    ncclFloat16    = ncclDataType(6), &
    ncclHalf       = ncclDataType(6), &
    ncclFloat32    = ncclDataType(7), &
    ncclFloat      = ncclDataType(7), &
    ncclFloat64    = ncclDataType(8), &
    ncclDouble     = ncclDataType(8), &
    ncclBfloat16   = ncclDataType(9), &
    ncclFloat8e4m3 = ncclDataType(10), &
    ncclFloat8e5m2 = ncclDataType(11), &
    ncclNumTypes   = ncclDataType(12)
```

```
! ncclRedOp
type, bind(c) :: ncclRedOp
  integer(c_int) :: member
end type ncclRedOp

type(ncclRedOp), parameter :: &
    ncclSum    = ncclRedOp(0), &
    ncclProd   = ncclRedOp(1), &
    ncclMax    = ncclRedOp(2), &
    ncclMin    = ncclRedOp(3), &
    ncclAvg    = ncclRedOp(4), &
    ncclNumOps = ncclRedOp(5)
```

The module provides a versioned derived type for each `ncclConfig_t` layout, so that fields added in successive NCCL releases can be reached without disturbing the earlier types. Choose the version that carries the configuration knobs you want to tune; `ncclConfig_v23100` has the full set.

```
! ncclConfig
type, bind(c) :: ncclConfig
  integer(c_size_t) :: size = 48
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = 21805
  integer(c_int) :: blocking = z'80000000'
  integer(c_int) :: cgaClusterSize = z'80000000'
  integer(c_int) :: minCTAs = z'80000000'
  integer(c_int) :: maxCTAs = z'80000000'
  type(c_ptr)    :: netName = c_null_ptr
  integer(c_int) :: splitShare = z'80000000'
end type ncclConfig

type, bind(c) :: ncclConfig_v22700
  integer(c_size_t) :: size = 72
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = NCCL_VERSION
  integer(c_int) :: blocking = z'80000000'
  integer(c_int) :: cgaClusterSize = z'80000000'
```

(continues on next page)

(continued from previous page)

```
  integer(c_int) :: minCTAs = z'80000000'
  integer(c_int) :: maxCTAs = z'80000000'
  type(c_ptr) :: netName = c_null_ptr
  integer(c_int) :: splitShare = z'80000000'
  integer(c_int) :: trafficClass = z'80000000'
  type(c_ptr) :: commName = c_null_ptr
  integer(c_int) :: collnetEnable = z'80000000'
  integer(c_int) :: CTAPolicy = z'80000000'
  integer(c_int) :: shrinkShare = z'80000000'
  integer(c_int) :: nvlsCTAs = z'80000000'
end type ncclConfig_v22700

type, bind(c) :: ncclConfig_v22800
  integer(c_size_t) :: size = 96
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = NCCL_VERSION
  integer(c_int) :: blocking = z'80000000'
  integer(c_int) :: cgaClusterSize = z'80000000'
  integer(c_int) :: minCTAs = z'80000000'
  integer(c_int) :: maxCTAs = z'80000000'
  type(c_ptr) :: netName = c_null_ptr
  integer(c_int) :: splitShare = z'80000000'
  integer(c_int) :: trafficClass = z'80000000'
  type(c_ptr) :: commName = c_null_ptr
  integer(c_int) :: collnetEnable = z'80000000'
  integer(c_int) :: CTAPolicy = z'80000000'
  integer(c_int) :: shrinkShare = z'80000000'
  integer(c_int) :: nvlsCTAs = z'80000000'
  integer(c_int) :: nChannelsPerNetPeer = z'80000000'
  integer(c_int) :: nvlinkCentricSched = z'80000000'
  integer(c_int) :: graphUsageMode = z'80000000'
  integer(c_int) :: numRmaCtx = z'80000000'
  integer(c_int) :: maxP2pPeers = z'80000000'
end type ncclConfig_v22800

type, bind(c) :: ncclConfig_v23100
  integer(c_size_t) :: size = 112
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = NCCL_VERSION
  integer(c_int) :: blocking = z'80000000'
  integer(c_int) :: cgaClusterSize = z'80000000'
  integer(c_int) :: minCTAs = z'80000000'
  integer(c_int) :: maxCTAs = z'80000000'
  type(c_ptr) :: netName = c_null_ptr
  integer(c_int) :: splitShare = z'80000000'
  integer(c_int) :: trafficClass = z'80000000'
  type(c_ptr) :: commName = c_null_ptr
  integer(c_int) :: collnetEnable = z'80000000'
  integer(c_int) :: CTAPolicy = z'80000000'
  integer(c_int) :: shrinkShare = z'80000000'
  integer(c_int) :: nvlsCTAs = z'80000000'
  integer(c_int) :: nChannelsPerNetPeer = z'80000000'
  integer(c_int) :: nvlinkCentricSched = z'80000000'
  integer(c_int) :: graphUsageMode = z'80000000'
  integer(c_int) :: numRmaCtx = z'80000000'
  integer(c_int) :: maxP2pPeers = z'80000000'
```

(continues on next page)

(continued from previous page)

```
  integer(c_int) :: graphStreamOrdering = z'80000000'
  integer(c_int) :: launchOrderImplicit = z'80000000'
  integer(c_int) :: numRmaSig = z'80000000'
  integer(c_int) :: rmaEagerInit = z'80000000'
  integer(c_int) :: hostCftMode = z'80000000'
end type ncclConfig_v23100
```

Renaming on use confines the version to a single line, which makes it easier to move to a later one:

```
use nccl, only : ncclConfig_t => ncclConfig_v23100
```

```
! ncclCollConfig (per-call configuration for the *Config collectives)
type, bind(c) :: ncclCollConfig
  integer(c_size_t) :: size = 64
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = NCCL_VERSION
  type(c_ptr) :: ext = c_null_ptr
  integer(c_int) :: minCTAs = z'80000000'
  integer(c_int) :: maxCTAs = z'80000000'
  integer(c_int) :: nvlsCTAs = z'80000000'
  integer(c_int) :: cgaClusterSize = z'80000000'
  type(c_ptr) :: algSelection = c_null_ptr
  integer(c_int) :: forceAlgSelection = 1
  integer(c_int) :: CTAPolicy = z'80000000'
  integer(c_int64_t) :: userProfilerTag = 0
end type ncclCollConfig

! ncclConfigExt (vendor-specific extension list, for the ext member of
! ncclCollConfig)
type, bind(c) :: ncclConfigExt
  type(c_ptr) :: next = c_null_ptr
  integer(c_int) :: vendorId = 0
  integer(c_int) :: optionId = 0
  type(c_ptr) :: val = c_null_ptr
end type ncclConfigExt
```

```
! ncclHostCftMode (for the hostCftMode member of ncclConfig_v23100)
type, bind(c) :: ncclHostCftMode
  integer(c_int) :: member
end type ncclHostCftMode

type(ncclHostCftMode), parameter :: &
    ncclHostCftDefault  = ncclHostCftMode(NCCL_CONFIG_UNDEF_INT), &
    ncclHostCftEnable   = ncclHostCftMode(1), &
    ncclHostCftDisable  = ncclHostCftMode(2), &
    ncclHostCftFallback = ncclHostCftMode(3)

! ncclParamHandle (opaque handle for a runtime parameter)
type, bind(c) :: ncclParamHandle
  type(c_ptr) :: member
end type ncclParamHandle
```

```
! ncclScalarResidence (for ncclRedOpCreatePreMulSum)
type, bind(c) :: ncclScalarResidence
  integer(c_int) :: member
end type ncclScalarResidence
```

(continues on next page)

(continued from previous page)

```
type(ncclScalarResidence), parameter :: &
    ncclScalarDevice        = ncclScalarResidence(0), &
    ncclScalarHostImmediate = ncclScalarResidence(1)
```

```
! ncclWindow (opaque handle for RMA windows)
type, bind(c) :: ncclWindow
  type(c_ptr) :: member
end type ncclWindow

! ncclWaitSignalDesc (for ncclWaitSignal)
type, bind(c) :: ncclWaitSignalDesc
  integer(c_int) :: opCnt
  integer(c_int) :: peer
  integer(c_int) :: sigIdx
  integer(c_int) :: ctx
end type ncclWaitSignalDesc
```

```
! ncclCommMemStat (for ncclCommMemStats)
type, bind(c) :: ncclCommMemStat
  integer(c_int) :: member
end type ncclCommMemStat

type(ncclCommMemStat), parameter :: &
    ncclStatGpuMemSuspend   = ncclCommMemStat(0), &
    ncclStatGpuMemSuspended = ncclCommMemStat(1), &
    ncclStatGpuMemPersist   = ncclCommMemStat(2), &
    ncclStatGpuMemTotal     = ncclCommMemStat(3)
```

```
! ncclSimInfo (for ncclGroupSimulateEnd)
type, bind(c) :: ncclSimInfo
  integer(c_size_t) :: size = 24
  integer(c_int) :: magic = z'74685283'
  integer(c_int) :: version = NCCL_VERSION
  real(c_float) :: estimatedTime = -1.0
end type ncclSimInfo
```

```
! ncclCommProperties (for ncclCommQueryProperties)
type, bind(c) :: ncclCommProperties
  integer(c_size_t) :: size = 56
  integer(c_int) :: magic = z'cafebeef'
  integer(c_int) :: version = NCCL_VERSION
  integer(c_int) :: rank = 0
  integer(c_int) :: nRanks = 0
  integer(c_int) :: cudaDev = 0
  integer(c_int) :: nvmlDev = 0
  logical(c_bool) :: deviceApiSupport = .false.
  logical(c_bool) :: multimemSupport = .false.
  integer(c_int) :: ginType = 0
  integer(c_int) :: nLsaTeams = 0
  logical(c_bool) :: hostRmaSupport = .false.
  integer(c_int) :: railedGinType = 0
end type ncclCommProperties
```

15.1.1. ncclGetLastError

This function returns a string for the last error that occurred.

```
character*128 function ncclGetLastError(comm)
type(ncclComm) :: comm
```

15.1.2. ncclGetErrorString

This function returns an error string for a given ncclResult value.

```
character*128 function ncclGetErrorString(ierr)
type(ncclResult) :: ierr
```

15.1.3. ncclGetVersion

This function returns the version number of the NCCL library.

```
type(ncclResult) function ncclGetVersion(version)
integer(4) :: version
```

15.1.4. ncclGetUniqueId

This function generates an ID to be used with ncclCommInitRank. This routine should be called once, and the generated ID should be distributed to all ranks.

```
type(ncclResult) function ncclGetUniqueId(uniqueId)
type(ncclUniqueId) :: uniqueId
```

15.1.5. ncclCommInitRank

This function generates a new NCCL communicator, of type(ncclCom). The rank argument must be between 0 and nranks-1. The uniqueId argument should be generated with ncclGetUniqueId.

```
type(ncclResult) function ncclCommInitRank(comm, nranks, uniqueId, rank)
type(ncclComm) :: comm
integer(4) :: nranks
type(ncclUniqueId) :: uniqueId
integer(4) :: rank
```

15.1.6. ncclCommInitAll

This function creates a single-process communicator clique, an array of type(ncclCom).

```
type(ncclResult) function ncclCommInitAll(comms, ndev, devlist)
type(ncclComm) :: comms(*)
integer(4) :: ndev
integer(4) :: devlist(*)
```

15.1.7. ncclCommInitRankConfig

This function generates a new NCCL communicator, of type(ncclCom), using a configuration argument. The rank argument must be between 0 and nranks-1. The uniqueId argument should be generated with ncclGetUniqueId. One variation of this interface will accept `c_null_ptr` for the last argument.

```
type(ncclResult) function ncclCommInitRankConfig(comm, nranks, &
       commId, rank, config)
type(ncclComm) :: comm
integer(4) :: nranks
type(ncclUniqueId) :: commId
integer(4) :: rank
type(ncclConfig) :: config ! or null, or type(ncclConfig_v22700),
                           ! or type(ncclConfig_v22800),
                           ! or type(ncclConfig_v23100)
```

15.1.8. ncclCommInitRankScalable

This function creates a communicator using multiple unique IDs for scalable initialization across multiple nodes. The config argument can be `c_null_ptr`, `type(ncclConfig_v22800)` or `type(ncclConfig_v23100)`.

```
type(ncclResult) function ncclCommInitRankScalable(newcomm, &
       nranks, myrank, nId, commIds, config)
type(ncclComm) :: newcomm
integer(4) :: nranks, myrank, nId
type(ncclUniqueId) :: commIds(*)
type(ncclConfig_v22800) :: config  ! or c_null_ptr,
                                   ! or type(ncclConfig_v23100)
```

15.1.9. ncclCommFinalize

This function finalizes the NCCL communicator object.

```
type(ncclResult) function ncclCommFinalize(comm)
type(ncclComm) :: comm
```

15.1.10. ncclCommDestroy

This function frees resources allocated to a NCCL communicator. It will wait for uncompleted operations.

```
type(ncclResult) function ncclCommDestroy(comm)
type(ncclComm) :: comm
```

15.1.11. ncclCommSplit

This function creates a new NCCL communicator, of type(ncclCom), from an existing one, based on the input color and key. A variation of this interface will accept `c_null_ptr` for the last argument.

```
type(ncclResult) function ncclCommSplit(comm, color, &
        key, newcomm, config)
type(ncclComm) :: comm
integer(4) :: color, key
type(ncclComm) :: newcomm
type(ncclConfig) :: config ! or null, or type(ncclConfig_v22700),
                           ! or type(ncclConfig_v22800),
                           ! or type(ncclConfig_v23100)
```

15.1.12. ncclCommShrink

This function creates a new communicator by excluding specified ranks from an existing communicator. Ranks listed in excludeRanksList are removed. The config argument can be `c_null_ptr`, `type(ncclConfig_v22800)` or `type(ncclConfig_v23100)`. The shrinkFlags argument should be `NCCL_SHRINK_DEFAULT` or `NCCL_SHRINK_ABORT`.

```
type(ncclResult) function ncclCommShrink(comm, excludeRanksList, &
        excludeRanksCount, newcomm, config, shrinkFlags)
type(ncclComm) :: comm
integer(4) :: excludeRanksList(*)
integer(4) :: excludeRanksCount
type(ncclComm) :: newcomm
type(ncclConfig_v22800) :: config  ! or c_null_ptr,
                                   ! or type(ncclConfig_v23100)
integer(4) :: shrinkFlags
```

15.1.13. ncclCommGetUniqueId

This function obtains a unique ID from an existing communicator, for use with ncclCommGrow to add new ranks to the communicator.

```
type(ncclResult) function ncclCommGetUniqueId(comm, uniqueId)
type(ncclComm) :: comm
type(ncclUniqueId), intent(out) :: uniqueId
```

15.1.14. ncclCommGrow

This function grows a communicator by adding new ranks. Existing ranks pass their current communicator; new ranks pass a null communicator and the uniqueId obtained from ncclCommGetUniqueId. The uniqueId argument can be `type(ncclUniqueId)` or `c_null_ptr` (for ranks that are neither rank 0 nor the joining rank). The config argument can be `c_null_ptr`, `type(ncclConfig_v22800)` or `type(ncclConfig_v23100)`. Must be called within ncclGroupStart/ncclGroupEnd.

```
type(ncclResult) function ncclCommGrow(comm, nRanks, uniqueId, &
        rank, newcomm, config)
type(ncclComm) :: comm
integer(4) :: nRanks
type(ncclUniqueId) :: uniqueId  ! or c_null_ptr
integer(4) :: rank
type(ncclComm) :: newcomm
type(ncclConfig_v22800) :: config  ! or c_null_ptr,
                                   ! or type(ncclConfig_v23100)
```

15.1.15. ncclCommRevoke

This function revokes a communicator, preventing further collective operations. After revocation, subsequent collective calls on the communicator return ncclInvalidUsage. The revokeFlags argument should be `NCCL_REVOKE_DEFAULT`.

```
type(ncclResult) function ncclCommRevoke(comm, revokeFlags)
type(ncclComm) :: comm
integer(4) :: revokeFlags
```

15.1.16. ncclCommAbort

This function frees resources allocated to a NCCL communicator. It will abort uncompleted operations.

```
type(ncclResult) function ncclCommAbort(comm)
type(ncclComm) :: comm
```

15.1.17. ncclCommRegister

This function registers a CUDA buffer, making it available for a zero-copy operation. The buffer can be of any type.

```
type(ncclResult) function ncclCommRegister(comm, buff, size, handle)
type(ncclComm) :: comm
real(4), device :: buff(*)  ! Any type of device array
integer(8)      :: size
type(c_ptr), intent(out) :: handle
```

15.1.18. ncclCommDeregister

This function deregisters a CUDA buffer used for a zero-copy operation.

```
type(ncclResult) function ncclCommDeregister(comm, handle)
type(ncclComm) :: comm
type(c_ptr), intent(in) :: handle
```

15.1.19. ncclCommWindowRegister

This function registers a device buffer as an RMA window. The buffer can be of any type. The winFlags argument controls window properties; use `NCCL_WIN_COLL_SYMMETRIC` for symmetric collective windows.

```
type(ncclResult) function ncclCommWindowRegister(comm, buff, &
    size, win, winFlags)
type(ncclComm) :: comm
real(4), device :: buff(*)  ! Any type of device array
integer(cuda_count_kind) :: size
type(ncclWindow) :: win
integer(4) :: winFlags
```

15.1.20. ncclCommWindowDeregister

This function deregisters a previously registered RMA window.

```
type(ncclResult) function ncclCommWindowDeregister(comm, win)
type(ncclComm) :: comm
type(ncclWindow) :: win
```

15.1.21. ncclWinGetUserPtr

This function returns the user pointer associated with a registered window.

```
type(ncclResult) function ncclWinGetUserPtr(comm, win, outUserPtr)
type(ncclComm) :: comm
type(ncclWindow) :: win
type(c_ptr), intent(out) :: outUserPtr
```

15.1.22. ncclCommGetAsyncError

This function queries whether the communicator has encountered any asynchronous errors.

```
type(ncclResult) function ncclCommGetAsyncError(comm, asyncError)
type(ncclComm) :: comm
type(ncclResult) :: asyncError
```

15.1.23. ncclCommCount

This function sets the count argument to the number of ranks in the NCCL communicator.

```
type(ncclResult) function ncclCommCount(comm, count)
type(ncclComm) :: comm
integer(4) :: count
```

15.1.24. ncclCommCuDevice

This function sets the device argument to the CUDA device associated with a NCCL communicator.

```
type(ncclResult) function ncclCommCuDevice(comm, device)
type(ncclComm) :: comm
integer(4) :: device
```

15.1.25. ncclCommUserRank

This function sets the rank argument to the rank within a NCCL communicator.

```
type(ncclResult) function ncclCommUserRank(comm, rank)
type(ncclComm) :: comm
integer(4) :: rank
```

15.1.26. ncclCommQueryProperties

This function queries the properties of a communicator, such as hardware support for device API, multicast, and host RMA. The props argument should be initialized with default values before calling.

```
type(ncclResult) function ncclCommQueryProperties(comm, props)
type(ncclComm) :: comm
type(ncclCommProperties) :: props
```

15.1.27. ncclMemAlloc

This function allocates a GPU buffer for use with NCCL.

```
type(ncclResult) function ncclMemAlloc(ptr, size)
type(c_devptr) :: ptr
integer(kind=cuda_count_kind) :: size
```

15.1.28. ncclMemFree

This function frees memory allocated with ncclMemAlloc.

```
type(ncclResult) function ncclMemFree(ptr)
type(c_devptr) :: ptr
```

15.1.29. ncclCommSuspend

This function suspends a communicator, optionally releasing GPU memory. Use `NCCL_SUSPEND_MEM` for the flags argument to release memory.

```
type(ncclResult) function ncclCommSuspend(comm, flags)
type(ncclComm) :: comm
integer(4) :: flags
```

15.1.30. ncclCommResume

This function resumes a previously suspended communicator.

```
type(ncclResult) function ncclCommResume(comm)
type(ncclComm) :: comm
```

15.1.31. ncclCommMemStats

This function queries GPU memory statistics for a communicator. The stat argument specifies which statistic to query (e.g. `ncclStatGpuMemTotal`).

```
type(ncclResult) function ncclCommMemStats(comm, stat, val)
type(ncclComm) :: comm
type(ncclCommMemStat) :: stat
integer(8), intent(out) :: val
```

15.2. NCCL Collective Communication Functions

This section contains interfaces for the NCCL functions that perform collective communication operations on device data. All functions can take either CUDA Fortran device arrays, OpenACC arrays within a host_data use_device data directive, or Fortran type(c_devptr) arguments. Each collective also has a `Config` variant that takes a `type(ncclCollConfig)` argument, which tunes how that one call is scheduled without changing the result it computes.

15.2.1. ncclAllReduce

This function performs the specified reduction on data across devices and writes the results into the receive buffer of every rank.

```
type(ncclResult) function ncclAllReduce(sendbuff, recvbuff, &
    count, datatype, op, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.2. ncclBroadcast

This function copies the send buffer on the root rank to all other ranks in the NCCL communicator. An in-place operation will happen if sendbuff and recvbuff are the same address.

```
type(ncclResult) function ncclBroadcast(sendbuff, recvbuff, &
    count, datatype, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.3. ncclReduce

This function performs the same operation as AllReduce, but writes the results only to the receive buffers of the specified root rank.

```
type(ncclResult) function ncclReduce(sendbuff, recvbuff, &
    count, datatype, op, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.4. ncclAllGather

This function gathers the send buffers from each rank and stores them in rank order in the receive buffer of all ranks.

```
type(ncclResult) function ncclAllGather(sendbuff, recvbuff, &
    sendcount, datatype, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: sendcount
type(ncclDataType) :: datatype
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.5. ncclReduceScatter

This function performs the specified reduction on the data, and leaves the result scattered in equal blocks among the ranks, based on the rank index.

```
type(ncclResult) function ncclReduceScatter(sendbuff, recvbuff, &
    recvcount, datatype, op, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
```

(continues on next page)

(continued from previous page)

```
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: recvcount
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.6. ncclAlltoAll

This function performs an all-to-all operation where each rank sends a distinct block of data to every other rank and receives a distinct block from every other rank. Each block contains count elements.

```
type(ncclResult) function ncclAlltoAll(sendbuff, recvbuff, &
    count, datatype, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.7. ncclGather

This function gathers sendcount elements from each rank's send buffer into the receive buffer of the root rank in rank order.

```
type(ncclResult) function ncclGather(sendbuff, recvbuff, &
    sendcount, datatype, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: sendcount
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.8. ncclScatter

This function scatters recvcount elements from the root rank's send buffer to each rank's receive buffer. The root's send buffer contains nranks blocks of recvcount elements in rank order.

```
type(ncclResult) function ncclScatter(sendbuff, recvbuff, &
    recvcount, datatype, root, comm, stream)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: recvcount
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.2.9. ncclAllReduceConfig

This function performs the same reduction as ncclAllReduce, using the supplied per-call configuration.

```
type(ncclResult) function ncclAllReduceConfig(sendbuff, &
    recvbuff, count, datatype, op, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.10. ncclBroadcastConfig

This function performs the same broadcast as ncclBroadcast, using the supplied per-call configuration.

```
type(ncclResult) function ncclBroadcastConfig(sendbuff, &
    recvbuff, count, datatype, root, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
```

(continues on next page)

(continued from previous page)

```
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.11. ncclReduceConfig

This function performs the same reduction as ncclReduce, using the supplied per-call configuration.

```
type(ncclResult) function ncclReduceConfig(sendbuff, recvbuff, &
    count, datatype, op, root, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.12. ncclAllGatherConfig

This function performs the same gather as ncclAllGather, using the supplied per-call configuration.

```
type(ncclResult) function ncclAllGatherConfig(sendbuff, &
    recvbuff, sendcount, datatype, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: sendcount
type(ncclDataType) :: datatype
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.13. ncclReduceScatterConfig

This function performs the same reduce-scatter as ncclReduceScatter, using the supplied per-call configuration.

```
type(ncclResult) function ncclReduceScatterConfig(sendbuff, &
    recvbuff, recvcount, datatype, op, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: recvcount
type(ncclDataType) :: datatype
type(ncclRedOp) :: op
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.14. ncclAlltoAllConfig

This function performs the same all-to-all exchange as ncclAlltoAll, using the supplied per-call configuration.

```
type(ncclResult) function ncclAlltoAllConfig(sendbuff, &
    recvbuff, count, datatype, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.15. ncclGatherConfig

This function performs the same gather as ncclGather, using the supplied per-call configuration.

```
type(ncclResult) function ncclGatherConfig(sendbuff, recvbuff, &
    count, datatype, root, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
```

(continues on next page)

(continued from previous page)

```
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.2.16. ncclScatterConfig

This function performs the same scatter as ncclScatter, using the supplied per-call configuration.

```
type(ncclResult) function ncclScatterConfig(sendbuff, recvbuff, &
    count, datatype, root, comm, stream, config)
type(c_devptr) :: sendbuff, recvbuff
  ! These combinations of sendbuff, recvbuff are also accepted:
  ! integer(4), device :: sendbuff(*), recvbuff(*)
  ! integer(8), device :: sendbuff(*), recvbuff(*)
  ! real(2), device :: sendbuff(*), recvbuff(*)
  ! real(4), device :: sendbuff(*), recvbuff(*)
  ! real(8), device :: sendbuff(*), recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: root
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
type(ncclCollConfig) :: config
```

15.3. NCCL Group Calls

This section contains interfaces for the NCCL functions that begin and end a group such that multiple calls can be merged.

15.3.1. ncclGroupStart

This function starts a group call. All subsequent calls to NCCL functions may not block due to inter-CPU synchronization.

```
type(ncclResult) function ncclGroupStart()
```

15.3.2. ncclGroupEnd

This function ends a group call. It returns when all operations since the corresponding call to ncclGroupStart have been processed, but not necessarily completed.

```
type(ncclResult) function ncclGroupEnd()
```

15.3.3. ncclGroupSimulateEnd

This function simulates ending a group call without actually executing it. It returns estimated timing information in the simInfo argument. The ncclSimInfo struct should be initialized with default values before calling.

```
type(ncclResult) function ncclGroupSimulateEnd(simInfo)
type(ncclSimInfo) :: simInfo
```

15.4. NCCL Point To Point Communication Functions

This section contains interfaces for the NCCL functions that perform point to point communication operations on device data. All functions can take either CUDA Fortran device arrays, OpenACC arrays within a host_data use_device data directive, or Fortran type(c_devptr) arguments.

15.4.1. Two-Sided Point-to-Point Operations

15.4.1.1 ncclSend

This function sends data from the send buffer to a communicator peer. This operation blocks the GPU. The receiving peer must call ncclRecv, with the same datatype and count.

```
type(ncclResult) function ncclSend(sendbuff, &
    count, datatype, peer, comm, stream)
type(c_devptr) :: sendbuff
  ! These types for sendbuff are also accepted:
  ! integer(4), device :: sendbuff(*)
  ! integer(8), device :: sendbuff(*)
  ! real(2), device :: sendbuff(*)
  ! real(4), device :: sendbuff(*)
  ! real(8), device :: sendbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: peer
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.4.1.2 ncclRecv

This function receives data from a communicator peer. This operation blocks the GPU. The sending peer must call ncclSend, with the same datatype and count.

```
type(ncclResult) function ncclRecv(recvbuff, &
    count, datatype, peer, comm, stream)
type(c_devptr) :: recvbuff
  ! These types for recvbuff are also accepted:
  ! integer(4), device :: recvbuff(*)
  ! integer(8), device :: recvbuff(*)
  ! real(2), device :: recvbuff(*)
  ! real(4), device :: recvbuff(*)
  ! real(8), device :: recvbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: peer
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.4.2. One-Sided Point-to-Point Operations (RMA)

The one-sided RMA functions require RMA-capable hardware (NVLink-connected GPUs) and communicators created with `numRmaCtx >= 1` in `ncclConfig_v22800`. Use `ncclCommQueryProperties` to check `hostRmaSupport` before calling these functions. Buffers must be allocated with `ncclMemAlloc` and registered as symmetric windows.

15.4.2.1 ncclPutSignal

This function performs a one-sided put operation, writing data from the local buffer into a remote peer's window and signaling completion. Must be called within ncclGroupStart/ncclGroupEnd.

```
type(ncclResult) function ncclPutSignal(localbuff, count, &
    datatype, peer, peerWin, peerWinOffset, sigIdx, ctx, &
    flags, comm, stream)
type(c_devptr) :: localbuff
  ! These types for localbuff are also accepted:
  ! integer(4), device :: localbuff(*)
  ! integer(8), device :: localbuff(*)
  ! real(2), device :: localbuff(*)
  ! real(4), device :: localbuff(*)
  ! real(8), device :: localbuff(*)
integer(cuda_count_kind) :: count
type(ncclDataType) :: datatype
integer(4) :: peer
type(ncclWindow) :: peerWin
integer(cuda_count_kind) :: peerWinOffset
integer(4) :: sigIdx
integer(4) :: ctx
integer(4) :: flags
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.4.2.2 ncclSignal

This function sends a signal to a remote peer without transferring data.

```
type(ncclResult) function ncclSignal(peer, sigIdx, ctx, flags, &
    comm, stream)
integer(4) :: peer
integer(4) :: sigIdx
integer(4) :: ctx
integer(4) :: flags
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.4.2.3 ncclWaitSignal

This function waits for one or more signals from remote peers, as described by the array of signal descriptors.

```
type(ncclResult) function ncclWaitSignal(nDesc, signalDescs, &
    comm, stream)
integer(4) :: nDesc
type(ncclWaitSignalDesc) :: signalDescs(*)
type(ncclComm) :: comm
integer(cuda_stream_kind) :: stream
```

15.5. User Defined Reduction Operators

This section contains interfaces for creating and destroying custom reduction operations.

15.5.1. ncclRedOpCreatePreMulSum

This function creates a custom pre-multiplied sum reduction operation. The scalar argument is a pointer to the scalar value, which can reside on the host (use `c_loc` with `ncclScalarHostImmediate`) or on the device (use `c_devloc` with `ncclScalarDevice`). The scalar type should match the datatype argument.

```
type(ncclResult) function ncclRedOpCreatePreMulSum(op, scalar, &
    datatype, residence, comm)
type(ncclRedOp), intent(out) :: op
type(c_ptr) :: scalar       ! host scalar via c_loc
  ! Also accepted: type(c_devptr) :: scalar  (device scalar via c_devloc)
type(ncclDataType) :: datatype
type(ncclScalarResidence) :: residence
type(ncclComm) :: comm
```

15.5.2. ncclRedOpDestroy

This function destroys a custom reduction operation previously created with ncclRedOpCreatePreMulSum.

```
type(ncclResult) function ncclRedOpDestroy(op, comm)
type(ncclRedOp) :: op
type(ncclComm) :: comm
```

15.6. NCCL Runtime Parameter Functions

This section contains interfaces for reading the NCCL runtime parameters, which are the settings NCCL takes from its `NCCL_*` environment variables. There are two styles. In the handle-based style, ncclParamBind looks a parameter up once and returns a type(ncclParamHandle) that is then read with one of the getters. In the key-based style, ncclParamGetParameter looks up a parameter and returns its value as a string in a single call.

15.6.1. ncclParamBind

This function binds the runtime parameter named by key to a handle that can be passed to the parameter getters. Valid key values are those returned by ncclParamGetAllParameterKeys. The handle is owned by the library and is not freed by the caller.

```
type(ncclResult) function ncclParamBind(h, key)
type(ncclParamHandle) :: h
character*(*) :: key
```

15.6.2. Typed Parameter Getters

These functions return the value of a bound parameter as an integer of a given width. Each one checks the type the parameter was registered with and returns `ncclInvalidArgument` on a mismatch, so exactly one of them succeeds for any given handle. Fortran has no unsigned integers, so the U forms are declared with the same-width signed kind; a U64 value larger than `huge(0_8)` reads back negative.

```
type(ncclResult) function ncclParamGetI8(h, val)
type(ncclParamHandle) :: h
integer(1) :: val

! The other seven getters differ only in the kind of val:
!   ncclParamGetI16, ncclParamGetU16 - integer(2) :: val
!   ncclParamGetI32, ncclParamGetU32 - integer(4) :: val
!   ncclParamGetI64, ncclParamGetU64 - integer(8) :: val
!   ncclParamGetU8                   - integer(1) :: val
```

15.6.3. ncclParamGetStr

This function returns the value of a bound parameter as a string. Two variations are provided. The character form copies the value into the caller's buffer, blank padded, and truncates if the buffer is short. The type(c_ptr) form returns the library's own pointer, which is valid only until the next ncclParamGetStr call on the same thread.

```
type(ncclResult) function ncclParamGetStr(h, val)
type(ncclParamHandle) :: h
character*(*) :: val ! or type(c_ptr) :: val
```

15.6.4. ncclParamGet

This function copies the raw bytes of a bound parameter into val, writing at most maxLen bytes and returning the number of bytes written in len. The val argument is host memory and may be of any type, kind or rank.

```
type(ncclResult) function ncclParamGet(h, val, maxLen, len)
type(ncclParamHandle) :: h
real(4) :: val(*) ! any type, kind or rank is accepted
integer(4) :: maxLen
integer(4) :: len
```

15.6.5. ncclParamGetParameter

This function looks up the parameter named by key and returns its value as a string, without binding a handle first. The valueLen argument returns the length the library reports, which may exceed `len(val)`; the copy into val is truncated in that case.

```
type(ncclResult) function ncclParamGetParameter(key, val, valueLen)
character*(*) :: key
character*(*) :: val
integer(4) :: valueLen
```

15.6.6. ncclParamGetAllParameterKeys

This function returns the library's table of parameter names. The table argument receives a pointer to an array of tableLen null-terminated C strings, owned by the library, which the caller walks with c_f_pointer. By default only the parameters published in the NCCL documentation are listed; setting `NCCL_PARAM_DUMP_ALL=1` in the environment includes the rest.

```
type(ncclResult) function ncclParamGetAllParameterKeys(table, tableLen)
type(c_ptr) :: table
integer(4) :: tableLen
```

15.6.7. ncclParamDumpAll

This subroutine writes every parameter in the registry, with its value, to the NCCL log output. By default only the parameters published in the NCCL documentation are written; setting NCCL_PARAM_DUMP_ALL=1 in the environment includes the rest.

```
subroutine ncclParamDumpAll()
```

Chapter 16. NVSHMEM Communication Library APIs

This section describes the Fortran interfaces to the NVSHMEM library. NVSHMEM is a software library that implements the OpenSHMEM application programming interface (API) for clusters of NVIDIA GPUs. OpenSHMEM is a community standard, one-sided communication API that provides a partitioned global address space (PGAS) parallel programming model. NVSHMEM provides an easy-to-use host-side interface for allocating symmetric memory, which can be distributed across a cluster of NVIDIA GPUs interconnected with NVLink, PCIe, and InfiniBand. The NVSHMEM communication functions are accessible from both host and device code. Most of the runtime API routines are written as C void functions, and we have implemented their Fortran wrappers as subroutines.

The NVSHMEM interfaces and definitions described in this chapter can be exposed by adding the line

```
use nvshmem
```

to your program unit. The same module is used for both host and device code. Device functions which run on a thread block or warp are declared as `acc vector nohost` routines. Others are `acc seq` routines.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4) and the plain real type implies real(4).

16.1. NVSHMEM Definitions, Setup, Exit, and Query Functions

This section contains definitions and data types used in the NVSHMEM library and interfaces to the NVSHMEM initialization and access to the parallel environment of the PEs.

The Fortran NVSHMEM module contains the following constant and derived type definitions:

```
! These are not available to the user, internal only
! defines, from nvshmemx_api.h
#define INIT_HANDLE_BYTES 128

! defines, from nvshmem_constants.h
#define SYNC_SIZE 27648
```

```
! Constant Definitions
integer, parameter :: NVSHMEM_SYNC_VALUE = 0
integer, parameter :: NVSHMEM_SYNC_SIZE = (2 * SYNC_SIZE)
integer, parameter :: NVSHMEM_BARRIER_SYNC_SIZE = (2 * SYNC_SIZE)
integer, parameter :: NVSHMEM_BCAST_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_REDUCE_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_REDUCE_MIN_WRKDATA_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_COLLECT_SYNC_SIZE = SYNC_SIZE
integer, parameter :: NVSHMEM_ALLTOALL_SYNC_SIZE = SYNC_SIZE

integer, parameter :: NVSHMEMX_CMP_EQ = 0
integer, parameter :: NVSHMEMX_CMP_NE = 1
integer, parameter :: NVSHMEMX_CMP_GT = 2
integer, parameter :: NVSHMEMX_CMP_LE = 3
integer, parameter :: NVSHMEMX_CMP_LT = 4
integer, parameter :: NVSHMEMX_CMP_GE = 5

integer, parameter :: NVSHMEMX_THREAD_SINGLE     = 0
integer, parameter :: NVSHMEMX_THREAD_FUNNELED   = 1
integer, parameter :: NVSHMEMX_THREAD_SERIALIZED = 2
integer, parameter :: NVSHMEMX_THREAD_MULTIPLE   = 3

integer, parameter :: NVSHMEM_TEAM_INVALID = -1
integer, parameter :: NVSHMEM_TEAM_WORLD   = 0
integer, parameter :: NVSHMEM_TEAM_SHARED  = 1
integer, parameter :: NVSHMEMX_TEAM_NODE               = 2
integer, parameter :: NVSHMEMX_TEAM_SAME_MYPE_NODE     = 3
integer, parameter :: NVSHMEMX_TEAM_MC_SHARED          = 6

integer, parameter :: NVSHMEM_MAJOR_VERSION = 1
integer, parameter :: NVSHMEM_MINOR_VERSION = 3

integer, parameter :: NVSHMEMX_INIT_THREAD_PES    = 1
integer, parameter :: NVSHMEMX_INIT_WITH_MPI_COMM = 2
integer, parameter :: NVSHMEMX_INIT_WITH_SHMEM    = 4
integer, parameter :: NVSHMEMX_INIT_WITH_HANDLE   = 8

! Region hints (NVSHMEM 3.8.0)
integer, parameter :: NVSHMEMX_REGION_HINT_NONE      = 0
integer, parameter :: NVSHMEMX_REGION_HINT_BATCH_RMA = 1
```

```
! Types from nvshmemx_api.h
type, bind(c) :: nvshmemx_init_handle
  character(c_char) :: content(INIT_HANDLE_BYTES)
end type nvshmemx_init_handle
```

```
! Types from nvshmemx_api.h
type, bind(c) :: nvshmemx_init_attr_type
  integer(8)  heap_size
  integer(4)  num_threads
  integer(4)  n_pes
  integer(4)  my_pe
  type(c_ptr) mpi_comm
  type(nvshmemx_init_handle) handle
end type nvshmemx_init_attr_type
```

```
! Types from nvshmem_types.h
type, bind(c) :: nvshmem_team_config
  integer(c_int) :: num_contexts
end type nvshmem_team_config
```

```
! Types from nvshmem_types.h (NVSHMEM 3.8.0)
! A region handle (nvshmemx_region_handle_t) is an integer(8)
type, bind(c) :: nvshmemx_region_attrs
  integer(4) :: hints
  character(c_char) :: reserved(60)
end type nvshmemx_region_attrs
```

```
! nvshmemx_status, from nvshmem_error.h
type, bind(c) :: nvshmemx_status
  integer(c_int) :: member
end type nvshmemx_status

type(nvshmemx_status), parameter :: &
  NVSHMEMX_SUCCESS = nvshmemx_status(0), &
  NVSHMEMX_ERROR_INVALID_VALUE = nvshmemx_status(1), &
  NVSHMEMX_ERROR_OUT_OF_MEMORY = nvshmemx_status(2), &
  NVSHMEMX_ERROR_NOT_SUPPORTED = nvshmemx_status(3), &
  NVSHMEMX_ERROR_SYMMETRY = nvshmemx_status(4), &
  NVSHMEMX_ERROR_GPU_NOT_SELECTED = nvshmemx_status(5), &
  NVSHMEMX_ERROR_COLLECTIVE_LAUNCH_FAILED = nvshmemx_status(6), &
  NVSHMEMX_ERROR_INTERNAL = nvshmemx_status(7)
```

16.1.1. nvshmem_init

This subroutine allocates and initializes resources used by the NVSHMEM library.

```
subroutine nvshmem_init()
```

16.1.2. nvshmemx_init_attr

This function initializes the NVSHMEM library based on an existing MPI communicator. Since the C and Fortran mpi_comm objects differ, this function has a different argument list than the corresponding C library entry point.

```
type(nvshmemx_status) function nvshmemx_init_attr(flags, comm)
    integer(4) :: flags, comm
```

Here is an example of using this function with MPI

```
use nvshmem
type(nvshmemx_status) :: nvstat
. . .
! Setup MPI
call MPI_Init(ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierror)
call MPI_Comm_size(MPI_COMM_WORLD, nranks, ierror)
```

(continues on next page)

(continued from previous page)

```
!
nvstat = nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, MPI_COMM_WORLD)
```

16.1.3. nvshmem_my_pe

This function returns the PE number of the calling PE, a number between 0 and npes-1.

```
integer(4) function nvshmem_my_pe()
```

16.1.4. nvshmem_n_pes

This function returns the number of PEs running in the program.

```
integer(4) function nvshmem_n_pes()
```

16.1.5. nvshmem_team_my_pe

This function returns the PE number of the calling PE, within the specified team.

```
integer(4) function nvshmem_team_my_pe(team)
    integer(4) :: team
```

16.1.6. nvshmem_team_n_pes

This function returns the number of PEs in the specified team.

```
integer(4) function nvshmem_team_n_pes(team)
    integer(4) :: team
```

16.1.7. nvshmem_team_get_config

This function returns the configuration parameters as described by the mask in the config argument.

```
integer(4) function nvshmem_team_get_config(team, mask, config)
    integer(4) :: team
    integer(8) :: mask
    type(nvshmem_team_config) :: config
```

16.1.8. nvshmem_team_translate_pe

This function returns the translated destination pe given the source team, the source pe, and the destination team.

```
integer(4) function nvshmem_team_translate_pe(src_team, src_pe, dest_team)
    integer(4) :: src_team, src_pe, dest_team
```

16.1.9. nvshmem_team_split_strided

This function performs a collective operation and creates a new team given a parent team and a desired slice (start, stride, and size) from the parent team.

```
integer(4) function nvshmem_team_split_strided(parent_team, &
        start, stride, size, config, mask, new_team)
    integer(4) :: parent_team
    integer(4) :: start, stride, size
    type(nvshmem_team_config) :: config
    integer(8) :: mask
    integer(4) :: new_team
```

16.1.10. nvshmem_team_split_2d

This function performs a collective operation and creates two new teams given a parent team and a specification of the 2D space. The result is two teams containing the PEs which map to the 2D space's row and column.

```
integer(4) function nvshmem_team_split_2d(parent_team, &
        xrange, xaxis_config, xaxis_mask, xaxis_team, &
        yaxis_config, yaxis_mask, yaxis_team)
    integer(4) :: parent_team
    integer(4) :: xrange
    type(nvshmem_team_config) :: xaxis_config, yaxis_config
    integer(8) :: xaxis_mask, yaxis_mask
    integer(4), intent(out) :: xaxis_team, yaxis_team
```

16.1.11. nvshmem_team_destroy

This function is a collective operation which destroys the team and frees the resouces associated with it.

```
integer(4) function nvshmem_team_destroy(team)
    integer(4) :: team
```

16.1.12. nvshmem_info_get_version

This subroutine returns the major and minor version number of the NVSHMEM library.

```
subroutine nvshmem_info_get_version(major, minor)
    integer(4) :: major, minor
```

16.1.13. nvshmem_info_get_name

This subroutine returns the vendor-defined name string for the library.

```
subroutine nvshmem_info_get_name(name)
    character*256, intent(out) :: name
```

16.1.14. nvshmem_finalize

This subroutine releases resources and ends the NVSHMEM portion of a program started with nvshmem_init().

```
subroutine nvshmem_finalize()
```

16.1.15. nvshmem_ptr

This function returns a local address that may be used to directly reference the destination data on the specified PE. The function nvshmem_ptr is implemented as a Fortran generic function, and can take any datatype, as long as it is a symmetric address.

```
type(c_devptr) function nvshmem_ptr(dest, pe)
    ! dest can be of type integer, logical, real, complex, character,
    ! or a type(c_devptr)
    integer(4) :: pe
```

The following specific functions are also supported:

```
type(c_devptr) function nvshmem_ptri(dest, pe)
    integer :: dest ! Any kind and rank
    integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrl(dest, pe)
    logical :: dest ! Any kind and rank
    integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrr(dest, pe)
    real :: dest ! Any kind and rank
    integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrc(dest, pe)
    complex :: dest ! Any kind and rank
    integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrc1(dest, pe)
    character :: dest ! Any kind and rank
    integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptrcd(dest, pe)
    type(c_devptr) :: dest
    integer(4) :: pe
```

16.2. NVSHMEM Memory Management Functions

This section contains the Fortran interfaces to NVSHMEM functions used to manage the symmetric heap.

16.2.1. nvshmem_malloc

This function allocates a block containing the specified number of bytes from the symmetric heap. This routine is a collective operation and requires participation by all PEs.

```
type(c_devptr) function nvshmem_malloc(size)
    integer(8) :: size  ! Size is in bytes
```

Entities of type(c_devptr) can be cast as Fortran arrays in a few ways. Here are some examples:

```
use nvshmem
! Contiguous will avoid some runtime checks
real(8), device, pointer, contiguous :: array(:)
. . .
call c_f_pointer(nvshmem_malloc(N*8), array, [N])
```

```
use nvshmem
! Cray Pointer
real(8), device :: array(N); pointer(pa,array)
. . .
pa = transfer(nvshmem_malloc(N*8), pa)
```

16.2.2. nvshmem_free

This subroutine frees a block of symmetric data which was previously allocated.

```
subroutine nvshmem_free(ptr)
    ! ptr can be of type(c_devptr), or other types if it was cast to a Fortran
    ! array using the techniques described in the nvshmem_malloc section.
```

16.2.3. nvshmem_align

This function allocates a block from the symmetric heap that has a byte alignment specified by the alignment argument.

```
type(c_devptr) function nvshmem_align(alignment, size)
    integer(8) :: alignment
    integer(8) :: size  ! Size is in bytes
```

16.2.4. nvshmem_calloc

This function allocates a block containing the specified number of bytes from the symmetric heap. This routine is a collective operation and requires participation by all PEs. The space is also initialized to zero.

```
type(c_devptr) function nvshmem_calloc(size)
    integer(8) :: size  ! Size is in bytes
```

16.2.5. nvshmemx_buffer_register

This function registers the given buffer with the remote transport and with CUDA for subsequent nvshmem operations. The address passed as the first argument can be of any type, with host or device attributes.

```
integer(4) function nvshmemx_buffer_register(addr, len)
    real(4) :: addr(*)  ! Any type is accepted
    integer(8) :: len   ! Size is in bytes
```

16.2.6. nvshmemx_buffer_unregister

This function unregisters the buffer which was previously registered with the nvshmemx_buffer_register library function.

```
integer(4) function nvshmemx_buffer_unregister(addr)
    real(4) :: addr(*)  ! Any type is accepted
```

16.2.7. nvshmemx_buffer_unregister_all

This subroutine unregisters all buffers which were previously registered with the nvshmemx_buffer_register library function.

```
subroutine nvshmemx_buffer_unregister_all()
```

16.3. NVSHMEM Remote Memory Access Functions

This section contains the Fortran interfaces to NVSHMEM functions used to perform reads and writes to symmetric data objects. The CUDA C library contains a number of functions for each C type. We have tried to distill those down into a useful, but non-redundant set for Fortran programmers. In addition, we have provided the following generic interfaces which are overloaded to take multiple types:

- nvshmem_put
- nvshmem_p
- nvshmem_iput
- nvshmem_put_nbi
- nvshmemx_put_block
- nvshmemx_put_warp
- nvshmem_get
- nvshmem_g
- nvshmem_iget
- nvshmem_get_nbi
- nvshmemx_get_block
- nvshmemx_get_warp

Many of these functions are available on both the host and device. Certain programming models may not currently support generic functions on the device. Some of the functions are only available on the device (most notably, those performed by a whole block or whole warp).

16.3.1. nvshmem_put

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine `nvshmem_put` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_putmem(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_putmem_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int8_put(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_put(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_put(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_put(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_put(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_put_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_put(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_put_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_put(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_on_stream(dest, source, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_put(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_on_stream(dest, source, nelems, pe, stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem put subroutines are not part of the generic nvshmem_put group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_put8(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put8_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put16(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put16_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put32(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put32_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put64(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put64_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put128(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put128_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.2. nvshmem_p

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine `nvshmem_p` is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device, and the source is passed by value and should be host-resident or device-resident, respectively. The specific names and argument lists are below.

```
subroutine nvshmem_int8_p(dest, source, nelems, pe)
    integer(1), device :: dest(*), source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_int8_p_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_int16_p(dest, source, nelems, pe)
    integer(2), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_int16_p_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_int32_p(dest, source, nelems, pe)
    integer(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_int32_p_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_int64_p(dest, source, nelems, pe)
    integer(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_int64_p_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_float_p(dest, source, nelems, pe)
    real(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_float_p_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_p(dest, source, nelems, pe)
    real(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_p_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.3. nvshmem_iput

This subroutine provides a way to copy strided data elements to a destination. This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine `nvshmem_iput` is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_iput(dest, source, dst, sst, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_iput(dest, source, dst, sst, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_iput(dest, source, dst, sst, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_iput(dest, source, dst, sst, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_iput(dest, source, dst, sst, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_iput(dest, source, dst, sst, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_iput(dest, source, dst, sst, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_iput_on_stream(dest, source, dst, sst, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_iput(dest, source, dst, sst, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_iput_on_stream(dest, source, dst, sst, nelems, pe,
↪stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem iput subroutines are not part of the generic nvshmem_iput group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_iput8(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iput8_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput16(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iput16_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput32(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iput32_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput64(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iput64_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iput128(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iput128_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.4. nvshmem_put_nbi

This subroutine returns after initiating the put operation. The subroutine `nvshmem_put_nbi` is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_put_nbi(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_put_nbi(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_put_nbi(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_put_nbi(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_put_nbi(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_put_nbi_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_put_nbi(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_put_nbi_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_put_nbi(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_nbi_on_stream(dest, source, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_put_nbi(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_nbi_on_stream(dest, source, nelems, pe, stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem put_nbi subroutines are not part of the generic nvshmem_put_nbi group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_put8_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put8_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put16_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put16_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put32_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put32_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put64_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put64_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_put128_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put128_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_putmem_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_putmem_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.5. nvshmemx_put_block

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine `nvshmemx_put_block` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_putmem_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_block(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_block(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_block(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_block(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_fp16_put_block(dest, source, nelems, pe)
    real(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_put_block(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_put_block(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_block(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_block(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

The following nvshmem put block subroutines are not part of the generic nvshmemx_put_block group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_put_block(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_long_put_block(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put8_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put16_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put32_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put64_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put128_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.6. nvshmemx_put_warp

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine `nvshmemx_put_warp` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_putmem_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_put_warp(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_put_warp(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_put_warp(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_put_warp(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_fp16_put_warp(dest, source, nelems, pe)
    real(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_put_warp(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_put_warp(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_put_warp(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_put_warp(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

The following nvshmem put warp subroutines are not part of the generic nvshmemx_put_warp group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_put_warp(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_long_put_warp(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put8_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put16_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put32_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put64_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_put128_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.7. nvshmemx_put_nbi_block

This subroutine initiates a non-blocking put operation and returns without waiting for completion. It is only available from device code as a block-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_nbi_block(dest, source, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_nbi_block group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_nbi_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.8. nvshmemx_put_nbi_warp

This subroutine initiates a non-blocking put operation and returns without waiting for completion. It is only available from device code as a warp-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_nbi_warp(dest, source, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_nbi_warp group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_nbi_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.9. nvshmemx_iput_block

This subroutine provides a way to copy strided data elements to a destination. It is only available from device code as a block-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_iput_block(dest, source, dst, sst, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

16.3.10. nvshmemx_iput_warp

This subroutine provides a way to copy strided data elements to a destination. It is only available from device code as a warp-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_iput_warp(dest, source, dst, sst, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

16.3.11. nvshmem_get

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine `nvshmem_get` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmem_getmem(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_getmem_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int8_get(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_get(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_get(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_get(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_get(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_get_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_get(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_get_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_get(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_on_stream(dest, source, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_get(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_on_stream(dest, source, nelems, pe, stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem get subroutines are not part of the generic nvshmem_get group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_get8(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_get8_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_get16(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_get16_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_get32(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_get32_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_get64(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_get64_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream

subroutine nvshmem_get128(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe

subroutine nvshmemx_get128_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.12. nvshmem_g

This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine nvshmem_g is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device, and the source is passed by value and should be host-resident or device-resident, respectively. The specific names and argument lists are below.

```
subroutine nvshmem_int8_g(dest, source, nelems, pe)
    integer(1), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_g_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_g(dest, source, nelems, pe)
    integer(2), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_g_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_g(dest, source, nelems, pe)
    integer(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_g_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_g(dest, source, nelems, pe)
    integer(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_g_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_g(dest, source, nelems, pe)
    real(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_g_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_g(dest, source, nelems, pe)
    real(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_g_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.13. nvshmem_iget

This subroutine provides a way to copy strided data elements to a destination. This subroutine returns after the data has been copied out of the source array on the local PE. The subroutine `nvshmem_iget` is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_iget(dest, source, dst, sst, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_iget(dest, source, dst, sst, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_iget(dest, source, dst, sst, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_iget(dest, source, dst, sst, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_iget(dest, source, dst, sst, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_iget(dest, source, dst, sst, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_iget(dest, source, dst, sst, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_iget_on_stream(dest, source, dst, sst, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_iget(dest, source, dst, sst, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_iget_on_stream(dest, source, dst, sst, nelems, pe,
↪stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem iget subroutines are not part of the generic nvshmem_iget group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_iget8(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iget8_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget16(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iget16_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget32(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iget32_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget64(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iget64_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_iget128(dest, source, dst, sst, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_iget128_on_stream(dest, source, dst, sst, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.14. nvshmem_get_nbi

This subroutine returns after initiating the get operation. The subroutine `nvshmem_get_nbi` is overloaded to take a number of different sets of arguments. These subroutines can be called from either the host or device. The specific names and argument lists are below.

```
subroutine nvshmem_int8_get_nbi(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int16_get_nbi(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int32_get_nbi(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_get_nbi(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_nbi_on_stream(dest, source, nelems, pe, stream)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_float_get_nbi(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_get_nbi_on_stream(dest, source, nelems, pe, stream)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_double_get_nbi(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_get_nbi_on_stream(dest, source, nelems, pe, stream)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_complex_get_nbi(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_nbi_on_stream(dest, source, nelems, pe, stream)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_dcomplex_get_nbi(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_nbi_on_stream(dest, source, nelems, pe, stream)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

The following nvshmem get_nbi subroutines are not part of the generic nvshmem_get_nbi group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmem_get8_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get8_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get16_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get16_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get32_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get32_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get64_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get64_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_get128_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get128_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_getmem_nbi(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_getmem_nbi_on_stream(dest, source, nelems, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
    integer(cuda_stream_kind) :: stream
```

16.3.15. nvshmemx_get_block

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine `nvshmemx_get_block` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_getmem_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_block(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_block(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_block(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_block(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_fp16_get_block(dest, source, nelems, pe)
    real(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_get_block(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_get_block(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_block(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_block(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

The following nvshmem get block subroutines are not part of the generic nvshmemx_get_block group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_get_block(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_long_get_block(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get8_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get16_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get32_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get64_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get128_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.16. nvshmemx_get_warp

This subroutine returns after the data has been copied out of the source array on the local PE. It is only available from device code. The subroutine `nvshmemx_get_warp` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
subroutine nvshmemx_getmem_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int8_get_warp(dest, source, nelems, pe)
    integer(1), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int16_get_warp(dest, source, nelems, pe)
    integer(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int32_get_warp(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_int64_get_warp(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_fp16_get_warp(dest, source, nelems, pe)
    real(2), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_float_get_warp(dest, source, nelems, pe)
    real(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_double_get_warp(dest, source, nelems, pe)
    real(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_complex_get_warp(dest, source, nelems, pe)
    complex(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_dcomplex_get_warp(dest, source, nelems, pe)
    complex(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

The following nvshmem get warp subroutines are not part of the generic nvshmemx_get_warp group, but are provided for flexibility and for compatibility with the C names:

```
subroutine nvshmemx_int_get_warp(dest, source, nelems, pe)
    integer(4), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_long_get_warp(dest, source, nelems, pe)
    integer(8), device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get8_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get16_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get32_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get64_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

```
subroutine nvshmemx_get128_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.17. nvshmemx_get_nbi_block

This subroutine initiates a non-blocking get operation and returns without waiting for completion. It is only available from device code as a block-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_get_nbi_block(dest, source, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_get_nbi_block group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_getmem_nbi_block(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.18. nvshmemx_get_nbi_warp

This subroutine initiates a non-blocking get operation and returns without waiting for completion. It is only available from device code as a warp-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_get_nbi_warp(dest, source, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_get_nbi_warp group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_getmem_nbi_warp(dest, source, nelems, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(4) :: pe
```

16.3.19. nvshmemx_iget_block

This subroutine provides a way to copy strided data elements from a source on a remote PE. It is only available from device code as a block-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_iget_block(dest, source, dst, sst, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

16.3.20. nvshmemx_iget_warp

This subroutine provides a way to copy strided data elements from a source on a remote PE. It is only available from device code as a warp-cooperative operation. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_iget_warp(dest, source, dst, sst, nelems, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: dst, sst, nelems
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

16.3.21. nvshmemx_put_signal_block

These subroutines perform a put operation followed by a signal update in a block-cooperative manner. They are only available from device code. The subroutine transfers `nelems` elements from `source` on the local PE to `dest` on PE `pe`, then writes `signal` to `sig_addr` on PE `pe` using the operation specified by `sig_op` (e.g., `NVSHMEM_SIGNAL_SET`). The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_block(dest, source, nelems, sig_addr, signal,
↪sig_op, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_block group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_block(dest, source, nelems, sig_addr, signal, sig_
↪op, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

16.3.22. nvshmemx_put_signal_warp

These subroutines perform a put operation followed by a signal update in a warp-cooperative manner. They are only available from device code. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_warp(dest, source, nelems, sig_addr, signal,
↪sig_op, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_warp group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_warp(dest, source, nelems, sig_addr, signal, sig_op,
↪ pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

16.3.23. nvshmemx_put_signal_on_stream

These subroutines perform a put-with-signal operation on a CUDA stream. They are callable only from the host. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_on_stream(dest, source, nelems, sig_addr,
↪signal, sig_op, pe, stream)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_on_stream group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_on_stream(dest, source, nelems, sig_addr, signal,
↪sig_op, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
    integer(cuda_stream_kind) :: stream
```

16.3.24. nvshmemx_put_signal_nbi_block

These subroutines initiate a non-blocking put-with-signal operation in a block-cooperative manner. They are only available from device code. They return after initiating the operation, without waiting for completion. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_nbi_block(dest, source, nelems, sig_addr,
↪signal, sig_op, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_nbi_block group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_nbi_block(dest, source, nelems, sig_addr, signal,
↪sig_op, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

16.3.25. nvshmemx_put_signal_nbi_warp

These subroutines initiate a non-blocking put-with-signal operation in a warp-cooperative manner. They are only available from device code. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_nbi_warp(dest, source, nelems, sig_addr,
↪signal, sig_op, pe)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_nbi_warp group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_nbi_warp(dest, source, nelems, sig_addr, signal,
↪sig_op, pe)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
```

16.3.26. nvshmemx_put_signal_nbi_on_stream

These subroutines initiate a non-blocking put-with-signal operation on a CUDA stream. They are callable only from the host. The specific names and argument lists are below.

```
subroutine nvshmemx_TYPENAME_put_signal_nbi_on_stream(dest, source, nelems, sig_addr,
↪signal, sig_op, pe, stream)
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

The following subroutine is not part of the generic nvshmemx_put_signal_nbi_on_stream group, but is provided for compatibility with the C names:

```
subroutine nvshmemx_putmem_signal_nbi_on_stream(dest, source, nelems, sig_addr,
↪signal, sig_op, pe, stream)
    type(c_devptr) :: dest, source
    integer(8) :: nelems
    integer(8), device :: sig_addr
    integer(8) :: signal
    integer(4) :: sig_op, pe
    integer(cuda_stream_kind) :: stream
```

16.4. NVSHMEM Collective Communication Functions

This section contains the Fortran interfaces to NVSHMEM functions that perform coordinated communication or synchronization operations within a group of PEs. The section can be further divided between barrier and sync functions, all-to-all, broadcast, and collect functions, and reductions.

16.4.1. nvshmem_barrier, nvshmem_barrier_all

These subroutines perform a collective synchronization over all (`nvshmem_barrier_all`) or a provided subset (`nvshmem_barrier`) of PEs. Ordering APIs initiated on the CPU only order communication operations that were issued from the CPU. Use cudaDeviceSynchronize() or something similar to ensure GPU operations have completed. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_barrier_all()
```

```
subroutine nvshmemx_barrier_all_on_stream(stream)
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_barrier(pe_start, pe_stride, pe_size, psync)
    integer(4) :: pe_start, pe_stride, pe_size
    integer(8), device :: psync(*)
```

```
subroutine nvshmemx_barrier_on_stream(pe_start, pe_stride, pe_size, psync, stream)
    integer(4) :: pe_start, pe_stride, pe_size
    integer(8), device :: psync(*)
    integer(cuda_stream_kind) :: stream
```

16.4.2. nvshmem_sync, nvshmem_sync_all

These subroutines perform a collective synchronization over all (`nvshmem_sync_all`) or a provided subset (`nvshmem_sync`) of PEs. Unlike the barrier routines, these subroutines only ensure completion and visibility of previously issued memory stores and does not ensure completion of remote memory updates. The list of subroutine names and argument lists are below.

```
subroutine nvshmem_sync_all()
```

```
subroutine nvshmemx_sync_all_on_stream(stream)
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_sync(pe_start, pe_stride, pe_size, psync)
    integer(4) :: pe_start, pe_stride, pe_size
    integer(8), device :: psync(*)
```

```
subroutine nvshmemx_sync_on_stream(pe_start, pe_stride, pe_size, psync, stream)
    integer(4) :: pe_start, pe_stride, pe_size
    integer(8), device :: psync(*)
    integer(cuda_stream_kind) :: stream
```

16.4.3. nvshmem_alltoall

These functions perform a collective all-to-all operation over a team. Starting in nvshmem version 2.0, the specific names for collective operations take a team argument and are specific to the type. These functions exchange the specified number of data elements with all other PEs in the team. These generic names are supported in the Fortran interfaces: nvshmem_alltoall, nvshmemx_alltoall_block, and nvshmemx_alltoall_warp. The nvshmem_alltoall functions are callable from host or device, the nvshmemx_alltoall_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_TYPENAME_alltoall(team, dest, source, nelems)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

```
integer function nvshmemx_TYPENAME_alltoall_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_alltoall_warp(team, dest, source, nelems)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nelems
```

where TYPENAME and TYPE are as in the table above.

16.4.4. nvshmem_broadcast

These functions perform a collective broadcast operation over a team. Starting in nvshmem version 2.0, the specific names for collective operations take a team argument and are specific to the type. These functions send the specified number of elements of source data from the specified root to all other PEs in the team. These generic names are supported in the Fortran interfaces: nvshmem_broadcast, nvshmemx_broadcast_block, and nvshmemx_broadcast_warp. The nvshmem_broadcast functions are callable from host or device, the nvshmemx_broadcast_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_broadcast(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmem_int16_broadcast(team, dest, source, nelems, pe_root)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmem_int32_broadcast(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmem_int64_broadcast(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmem_float_broadcast(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmem_double_broadcast(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
```

```
integer function nvshmemx_int8_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_broadcast_on_stream(team, &
        dest, source, nelems, pe_root, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int16_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int32_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int64_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_float_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_double_broadcast_block(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
```

```
integer function nvshmemx_int8_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int16_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int32_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_int64_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_float_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root

integer function nvshmemx_double_broadcast_warp(team, dest, source, nelems, pe_root)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
    integer(4) :: pe_root
```

16.4.5. nvshmem_collect

These functions perform a collective operation to concatenate the specified number of elements from each source array into the dest array for each PE in the team. Starting in nvshmem version 2.0, the specific names for collective operations take a team argument and are specific to the type. The collected data is in order of the PE in the team, and nelems can vary from PE to PE. These generic names are supported in the Fortran interfaces: nvshmem_collect, nvshmemx_collect_block, and nvshmemx_collect_warp. The nvshmem_collect functions are callable from host or device, the nvshmemx_collect_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_collect(team, dest, source, nelems)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems

integer function nvshmem_int16_collect(team, dest, source, nelems)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(2), device :: dest, source
    integer(8) :: nelems

integer function nvshmem_int32_collect(team, dest, source, nelems)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmem_int64_collect(team, dest, source, nelems)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems

integer function nvshmem_float_collect(team, dest, source, nelems)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmem_double_collect(team, dest, source, nelems)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
```

```
integer function nvshmemx_int8_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
```

(continues on next page)

(continued from previous page)

```
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_collect_on_stream(team, &
        dest, source, nelems, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_collect_block(team, dest, source, nelems)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int16_collect_block(team, dest, source, nelems)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int32_collect_block(team, dest, source, nelems)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int64_collect_block(team, dest, source, nelems)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_float_collect_block(team, dest, source, nelems)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_double_collect_block(team, dest, source, nelems)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
```

```
integer function nvshmemx_int8_collect_warp(team, dest, source, nelems)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int16_collect_warp(team, dest, source, nelems)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int32_collect_warp(team, dest, source, nelems)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int64_collect_warp(team, dest, source, nelems)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(8), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_float_collect_warp(team, dest, source, nelems)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_double_collect_warp(team, dest, source, nelems)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nelems
```

16.4.6. NVSHMEM Reductions

This section contains the Fortran interfaces to NVSHMEM functions that perform reductions, which are synchronization operations within a group of PEs performing a bitwise or arithmetic operation, reducing a set of values down to one.

16.4.6.1 nvshmem_and_reduce

These functions perform a bitwise AND reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for the reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_and_reduce, nvshmemx_and_reduce_block, and nvshmemx_and_reduce_warp. The nvshmem_and_reduce functions are callable from host or device, the nvshmemx_and_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_and_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_and_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_and_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_and_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_and_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_and_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_and_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_and_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_and_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_and_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_and_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_and_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_and_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_and_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_and_reduce_warp(team, dest, source, nreduce)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_and_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.2 nvshmem_or_reduce

These functions perform a bitwise OR reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for the reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_or_reduce, nvshmemx_or_reduce_block, and nvshmemx_or_reduce_warp. The nvshmem_or_reduce functions are callable from host or device, the nvshmemx_or_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_or_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_or_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_or_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_or_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_or_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_or_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_or_reduce_on_stream(team, &
        dest, source, nreduce, stream)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_or_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_or_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_or_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_or_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_or_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_or_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_or_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_or_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_or_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.3 nvshmem_xor_reduce

These functions perform a bitwise XOR reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for the reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_xor_reduce, nvshmemx_xor_reduce_block, and nvshmemx_xor_reduce_warp. The nvshmem_xor_reduce functions are callable from host or device, the nvshmemx_xor_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_xor_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_xor_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_xor_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_xor_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_xor_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_xor_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_xor_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_xor_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_xor_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_xor_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_xor_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_xor_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_xor_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_xor_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_xor_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_xor_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.4 nvshmem_max_reduce

These functions perform a maximum value, MAX, reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_max_reduce, nvshmemx_max_reduce_block, and nvshmemx_max_reduce_warp. The nvshmem_max_reduce functions are callable from host or device, the nvshmemx_max_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_max_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_max_reduce(team, dest, source, nreduce)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_max_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_max_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_float_max_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_double_max_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(4), device :: dest, source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_max_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_max_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_max_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_max_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_max_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_max_reduce_warp(team, dest, source, nreduce)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_max_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_max_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.5 nvshmem_min_reduce

These functions perform a minimum value, MIN, reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_min_reduce, nvshmemx_min_reduce_block, and nvshmemx_min_reduce_warp. The nvshmem_min_reduce functions are callable from host or device, the nvshmemx_min_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_float_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_double_min_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_min_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
```

(continues on next page)

(continued from previous page)

```
integer function nvshmemx_int64_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_min_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_min_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.6 nvshmem_sum_reduce

These functions perform a summation, or SUM, reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_sum_reduce, nvshmemx_sum_reduce_block, and nvshmemx_sum_reduce_warp. The nvshmem_sum_reduce functions are callable from host or device, the nvshmemx_sum_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_float_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_double_sum_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
```

(continues on next page)

(continued from previous page)

```
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_sum_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_sum_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_sum_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.6.7 nvshmem_prod_reduce

These functions perform a product reduction across a set of PEs in a team. Starting in nvshmem version 2.0, the specific names for reduction operations take a team argument and are specific to the type. These generic names are supported in the Fortran interfaces: nvshmem_prod_reduce, nvshmemx_prod_reduce_block, and nvshmemx_prod_reduce_warp. The nvshmem_prod_reduce functions are callable from host or device, the nvshmemx_prod_reduce_on_stream functions are callable only from the host, and the block and warp functions are callable only from the device.

```
integer function nvshmem_int8_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int16_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int32_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_int64_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_float_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmem_double_prod_reduce(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int16_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int32_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_int64_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_float_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_prod_reduce_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

```
integer function nvshmemx_int8_prod_reduce_block(team, dest, source, nreduce)
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_prod_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_prod_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_prod_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_prod_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_prod_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    real(8), device :: dest, source
    integer(8) :: nreduce
```

```
integer function nvshmemx_int8_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int16_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int32_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_int64_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_float_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nreduce

integer function nvshmemx_double_prod_reduce_warp(team, dest, source, nreduce)
    integer(4) :: team
```

(continues on next page)

(continued from previous page)

```
    real(8), device :: dest, source
    integer(8) :: nreduce
```

16.4.7. NVSHMEM Reduce-Scatter

This section contains the Fortran interfaces to NVSHMEM functions that perform reduce-scatter operations. Each PE's source buffer contains `nreduce * team_size` elements (one `nreduce`-element block per PE in the team); after the operation each PE's destination buffer holds `nreduce` reduced elements. The `nvshmem_*_reducescatter` functions are callable from host or device, the `nvshmemx_*_reducescatter_on_stream` functions are callable only from the host, and the block and warp functions are callable only from the device.

16.4.7.1 nvshmem_and_reducescatter

These functions perform a bitwise AND reduce-scatter across a set of PEs in a team. The generic names supported in the Fortran interfaces are: `nvshmem_and_reducescatter`, `nvshmemx_and_reducescatter_block`, and `nvshmemx_and_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_and_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the integer RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)

```
integer function nvshmemx_TYPENAME_and_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_and_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_and_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.2 nvshmem_or_reducescatter

These functions perform a bitwise OR reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_or_reducescatter`, `nvshmemx_or_reducescatter_block`, and `nvshmemx_or_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_or_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the integer RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)

```
integer function nvshmemx_TYPENAME_or_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_or_reducescatter_block(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_or_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.3 nvshmem_xor_reducescatter

These functions perform a bitwise XOR reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_xor_reducescatter`, `nvshmemx_xor_reducescatter_block`, and `nvshmemx_xor_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_xor_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the integer RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)

```
integer function nvshmemx_TYPENAME_xor_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_xor_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_xor_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.4 nvshmem_max_reducescatter

These functions perform a maximum reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_max_reducescatter`, `nvshmemx_max_reducescatter_block`, and `nvshmemx_max_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_max_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

```
integer function nvshmemx_TYPENAME_max_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_max_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_max_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.5 nvshmem_min_reducescatter

These functions perform a minimum reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_min_reducescatter`, `nvshmemx_min_reducescatter_block`, and `nvshmemx_min_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_min_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

```
integer function nvshmemx_TYPENAME_min_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_min_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_min_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.6 nvshmem_sum_reducescatter

These functions perform a sum reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_sum_reducescatter`, `nvshmemx_sum_reducescatter_block`, and `nvshmemx_sum_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_sum_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

```
integer function nvshmemx_TYPENAME_sum_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_sum_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_sum_reducescatter_warp(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.7 nvshmem_prod_reducescatter

These functions perform a product reduce-scatter across a set of PEs in a team. The generic names supported are: `nvshmem_prod_reducescatter`, `nvshmemx_prod_reducescatter_block`, and `nvshmemx_prod_reducescatter_warp`.

```
integer function nvshmem_TYPENAME_prod_reducescatter(team, dest, source, nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are one of the standard RMA types and names:

TYPENAME	TYPE
int8	integer(1)
int16	integer(2)
int32	integer(4)
int64	integer(8)
float	real(4)
double	real(8)

```
integer function nvshmemx_TYPENAME_prod_reducescatter_on_stream(team, &
        dest, source, nreduce, stream)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
    integer(cuda_stream_kind) :: stream
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_prod_reducescatter_block(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

```
integer function nvshmemx_TYPENAME_prod_reducescatter_warp(team, dest, source,
↪nreduce)
    integer(4) :: team
    TYPE, device :: dest(*), source(*)
    integer(8) :: nreduce
```

where TYPENAME and TYPE are as in the table above.

16.4.7.8 nvshmemx_double2_maxloc_reduce_block

This function performs a double2 max-location reduction across a set of PEs in a team as a block-cooperative operation. It is only available from device code. The C `double2` type maps to Fortran `complex(8)` (same memory layout: two consecutive `real(8)` values). The first element holds the value; the second holds the location index used to determine the maximum.

Note: This implementation requires `nreduce == 1`.

```
integer function nvshmemx_double2_maxloc_reduce_block(team, dest, source, nreduce)
    integer(4) :: team
    complex(8), device :: dest(*), source(*)
    integer(8) :: nreduce
```

16.5. NVSHMEM Atomic Memory Operations

This section contains the Fortran interfaces to NVSHMEM atomic memory operations (AMOs). These operations provide atomic read-modify-write access to symmetric memory objects. The `nvshmem_TYPENAME_*` functions are callable from both host and device code.

16.5.1. nvshmem_atomic_inc

This subroutine performs an atomic increment of the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_inc(dest, pe)
    TYPE, device :: dest(*)
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the integer AMO types and names:

TYPENAME	TYPE
int32	integer(4)
int64	integer(8)

16.5.2. nvshmem_atomic_fetch_inc

This function atomically increments the symmetric variable `dest` on PE `pe` and returns its old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch_inc(dest, pe)
    TYPE, device :: dest(*)
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.3. nvshmem_atomic_fetch

This function atomically fetches the value of the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch(dest, pe)
    TYPE, device :: dest(*)
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.4. nvshmem_atomic_set

This subroutine atomically sets the symmetric variable `dest` on PE `pe` to `val`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_set(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.5. nvshmem_atomic_add

This subroutine atomically adds `val` to the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_add(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.6. nvshmem_atomic_fetch_add

This function atomically adds `val` to the symmetric variable `dest` on PE `pe` and returns its old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch_add(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.7. nvshmem_atomic_swap

This function atomically swaps the symmetric variable `dest` on PE `pe` with `val` and returns the old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_swap(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.8. nvshmem_atomic_compare_swap

This function atomically compares `dest` on PE `pe` with `cond`; if equal it replaces `dest` with `val` and returns the old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_compare_swap(dest, cond, val, pe)
    TYPE, device :: dest(*)
    TYPE :: cond
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.9. nvshmem_atomic_and

This subroutine atomically ANDs `val` into the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_and(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.10. nvshmem_atomic_fetch_and

This function atomically ANDs `val` into the symmetric variable `dest` on PE `pe` and returns the old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch_and(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.11. nvshmem_atomic_or

This subroutine atomically ORs `val` into the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_or(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.12. nvshmem_atomic_fetch_or

This function atomically ORs `val` into the symmetric variable `dest` on PE `pe` and returns the old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch_or(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.13. nvshmem_atomic_xor

This subroutine atomically XORs `val` into the symmetric variable `dest` on PE `pe`. Callable from host and device.

```
subroutine nvshmem_TYPENAME_atomic_xor(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.14. nvshmem_atomic_fetch_xor

This function atomically XORs `val` into the symmetric variable `dest` on PE `pe` and returns the old value. Callable from host and device.

```
TYPE function nvshmem_TYPENAME_atomic_fetch_xor(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.5.15. nvshmemx_atomic_add

This subroutine atomically adds `val` to the symmetric variable `dest` on PE `pe` for floating-point types. Callable from device code only.

```
subroutine nvshmemx_TYPENAME_atomic_add(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are one of the floating-point AMO types and names:

TYPENAME	TYPE
float	real(4)
double	real(8)

16.5.16. nvshmemx_atomic_fetch_add

This function atomically adds `val` to the symmetric variable `dest` on PE `pe` for floating-point types and returns the old value. Callable from device code only.

```
TYPE function nvshmemx_TYPENAME_atomic_fetch_add(dest, val, pe)
    TYPE, device :: dest(*)
    TYPE :: val
    integer(4) :: pe
```

where TYPENAME and TYPE are as in the table above.

16.6. NVSHMEM Point to Point Synchronization Functions

This section contains the Fortran interfaces to NVSHMEM functions that provide a mechanism for synchronization between two PEs based on a value in the symmetric memory.

16.6.1. nvshmem_wait_until

This subroutine blocks until the value contained in the symmetric data object at the calling PE satisfies the condition specified by the comparison operator and the comparison value. The subroutine `nvshmem_wait_until` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below. The variants that take a stream are available from host code. The variants that do not take a stream argument are device code only.

```
subroutine nvshmem_int32_wait_until(ivar, cmp, cvalue)
    integer(4), device :: ivar
    integer(4) :: cmp, cvalue
```

```
subroutine nvshmemx_int32_wait_until_on_stream(ivar, cmp, cvalue, stream)
    integer(4), device :: ivar
    integer(4) :: cmp, cvalue
    integer(cuda_stream_kind) :: stream
```

```
subroutine nvshmem_int64_wait_until(ivar, cmp, cvalue)
    integer(8), device :: ivar
    integer(4) :: cmp
    integer(8) :: cvalue
```

```
subroutine nvshmemx_int64_wait_until_on_stream(ivar, cmp, cvalue, stream)
    integer(8), device :: ivar
    integer(4) :: cmp
    integer(8) :: cvalue
    integer(cuda_stream_kind) :: stream
```

16.6.2. nvshmem_wait_until_all

These device subroutines block until all values contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The subroutine `nvshmem_wait_until_all` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
subroutine nvshmem_int32_wait_until_all(ivars, nelems, status, cmp, cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
subroutine nvshmem_int64_wait_until_all(ivars, nelems, status, cmp, cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.3. nvshmem_wait_until_any

These device functions block until any one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The value returned is the index in the array that satisfies the condition. The function `nvshmem_wait_until_any` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_wait_until_any(ivars, nelems, status, cmp, cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_wait_until_any(ivars, nelems, status, cmp, cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.4. nvshmem_wait_until_some

These device functions wait until at least one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The indices array, upon return, contains those indices which satisfy the condition. The function return value is the number of elements that satisfied the condition. The function `nvshmem_wait_until_some` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_wait_until_some(ivars, nelems, indices, status, cmp,
↪ cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_wait_until_some(ivars, nelems, indices, status, cmp,
↪ cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.5. nvshmem_wait_until_all_vector

These device subroutines block until all values contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The subroutine `nvshmem_wait_until_all_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
subroutine nvshmem_int32_wait_until_all_vector(ivars, nelems, status, cmp, cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
subroutine nvshmem_int64_wait_until_all_vector(ivars, nelems, status, cmp, cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
```

(continues on next page)

(continued from previous page)

```
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.6. nvshmem_wait_until_any_vector

These device functions block until any one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The value returned is the index in the array that satisfies the condition. The function `nvshmem_wait_until_any_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_wait_until_any_vector(ivars, nelems, status, cmp,
↪cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_wait_until_any_vector(ivars, nelems, status, cmp,
↪cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.7. nvshmem_wait_until_some_vector

These device functions wait until at least one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The indices array, upon return, contains those indices which satisfy the condition. The function return value is the number of elements that satisfied the condition. The function `nvshmem_wait_until_some_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_wait_until_some_vector(ivars, nelems, indices,
↪status, cmp, cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_wait_until_some_vector(ivars, nelems, indices,
↪status, cmp, cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.8. nvshmem_test

These device functions test whether the value contained in the symmetric data object at the calling PE satisfies the condition specified by the comparison operator and the comparison value. The functions return “1” if the comparison evaluates to true, “0” otherwise. The function `nvshmem_test` is overloaded to take a number of different sets of arguments. The specific names and argument lists are below.

```
integer(4) function nvshmem_int32_test(ivar, cmp, cvalue)
    integer(4), device :: ivar, cvalue
    integer(4) :: cmp
```

```
integer(4) function nvshmem_int64_test(ivar, cmp, cvalue)
    integer(8), device :: ivar, cvalue
    integer(4) :: cmp
```

16.6.9. nvshmem_test_all

These device functions test all values contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The functions return “1” if all comparisons evaluate to true, “0” otherwise. The function `nvshmem_test_all` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(4) nvshmem_int32_test_all(ivars, nelems, status, cmp, cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(4) nvshmem_int64_test_all(ivars, nelems, status, cmp, cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.10. nvshmem_test_any

These device functions test whether any value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The functions return “1” if any comparisons evaluate to true, “0” otherwise. The value returned is the index in the array that satisfies the condition. The function `nvshmem_test_any` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_test_any(ivars, nelems, status, cmp, cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_test_any(ivars, nelems, status, cmp, cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.11. nvshmem_test_some

These device functions test whether at least one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value. The indices array, upon return, contains those indices which satisfy the condition. The function return value is the number of elements that satisfied the condition. The function `nvshmem_test_some` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_test_some(ivars, nelems, indices, status, cmp,
↪cvalue)
    integer(4), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_test_some(ivars, nelems, indices, status, cmp,
↪cvalue)
    integer(8), device :: ivars(*), cvalue
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.12. nvshmem_test_all_vector

These device functions test whether all values contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The functions return “1” if all comparisons evaluate to true, “0” otherwise. The function `nvshmem_test_all_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
subroutine nvshmem_int32_test_all_vector(ivars, nelems, status, cmp, cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
subroutine nvshmem_int64_test_all_vector(ivars, nelems, status, cmp, cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.13. nvshmem_test_any_vector

These device functions test wheter any one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The value returned is the index in the array that satisfies the condition. The function `nvshmem_test_any_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_test_any_vector(ivars, nelems, status, cmp, cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_test_any_vector(ivars, nelems, status, cmp, cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.14. nvshmem_test_some_vector

These device functions test whether at least one value contained in the wait set at the calling PE satisfy the condition specified by the comparison operator and the comparison value vector. The indices array, upon return, contains those indices which satisfy the condition. The function return value is the number of elements that satisfied the condition. The function `nvshmem_test_some_vector` is overloaded to take two different sets of integer arguments. The specific names and argument lists are below.

```
integer(8) function nvshmem_int32_test_some_vector(ivars, nelems, indices, status,
↪cmp, cvalues)
    integer(4), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

```
integer(8) function nvshmem_int64_test_some_vector(ivars, nelems, indices, status,
↪cmp, cvalues)
    integer(8), device :: ivars(*), cvalues(*)
    integer(8) :: nelems
    integer(8), intent(out) :: indices(nelems)
    integer(4) :: status(*)
    integer(4) :: cmp
```

16.6.15. nvshmemx_signal_counted_reset

This subroutine resets the signal word at `sig_addr` on the calling PE to zero. It is callable from host or device code. Only `nvshmemx_signal_counted_reset` is available in Fortran; the other counted-signal functions in NVSHMEM 3.8.0 (`nvshmemx_signal_counted_load`, `nvshmemx_signal_counted_wait_until`, and `nvshmemx_putmem_signal_counted_nbi_block`) are only declared by NVSHMEM builds that enable `NVSHMEM_CFT_HANDLES_SUPPORT`, which the NVSHMEM libraries shipped with the NVIDIA HPC SDK do not.

```
subroutine nvshmemx_signal_counted_reset(sig_addr)
    integer(8), device :: sig_addr
```

16.7. NVSHMEM Memory Ordering Functions

This section contains the Fortran interfaces to NVSHMEM functions that provide mechanisms to ensure ordering and/or delivery of completion on NVSHMEM operations. Beginning with the 25.3 release, nvshmem_fence and nvshmem_quiet are also callable from Fortran device code.

16.7.1. nvshmem_fence

This subroutine ensures the ordering of delivery of operations on symmetric data objects.

```
subroutine nvshmem_fence()
```

16.7.2. nvshmem_quiet

These subroutines ensure completion of all operations on symmetric data objects issued by the calling PE.

```
subroutine nvshmem_quiet()
```

```
subroutine nvshmemx_quiet_on_stream(stream)
    integer(cuda_stream_kind) :: stream
```

16.7.3. nvshmemx_flush

These subroutines provide a source-buffer-reuse guarantee: after `nvshmemx_flush` returns, the caller's put source buffers are safe to reuse. Unlike `nvshmem_quiet`, remote visibility is not guaranteed. `nvshmemx_flush` is callable from host or device (single thread); `nvshmemx_flush_warp` and `nvshmemx_flush_block` are device-only cooperative variants; `nvshmemx_flush_on_stream` is host-only and stream-ordered.

```
subroutine nvshmemx_flush()
```

```
subroutine nvshmemx_flush_warp()
```

```
subroutine nvshmemx_flush_block()
```

```
subroutine nvshmemx_flush_on_stream(stream)
    integer(cuda_stream_kind) :: stream
```

16.8. NVSHMEM Region Functions

This section contains the Fortran interfaces to the NVSHMEM region functions, introduced in NVSHMEM 3.8.0. A region brackets a sequence of non-blocking remote memory access operations and lets the library apply the optimizations requested by the `hints` field of `nvshmemx_region_attrs`. The only hint currently defined besides `NVSHMEMX_REGION_HINT_NONE` is `NVSHMEMX_REGION_HINT_BATCH_RMA`. `nvshmemx_region_start` and `nvshmemx_region_stop` are called from host code; `nvshmemx_region_start_block` and `nvshmemx_region_stop_block` are device-only and must be called by all threads of a thread block. All of these functions return zero on success and a nonzero `nvshmemx_status` value on failure. The `reserved` member of `nvshmemx_region_attrs` must be set to zero.

16.8.1. nvshmemx_region_start, nvshmemx_region_stop

These functions start and stop a region from host code. `nvshmemx_region_start` returns a region `handle` that is passed to `nvshmemx_region_stop`.

```
integer(4) function nvshmemx_region_start(handle, attrs)
    integer(8) :: handle
    type(nvshmemx_region_attrs) :: attrs
```

```
integer(4) function nvshmemx_region_stop(handle)
    integer(8), value :: handle
```

16.8.2. nvshmemx_region_start_block, nvshmemx_region_stop_block

These functions start and stop a region from device code, for the calling thread block. They are only available from device code, and must be called by all threads of the block.

```
attributes(device) integer(4) function nvshmemx_region_start_block(handle, attrs)
    integer(8) :: handle
    type(nvshmemx_region_attrs) :: attrs
```

```
attributes(device) integer(4) function nvshmemx_region_stop_block(handle)
    integer(8), value :: handle
```

16.8.3. nvshmemx_region_is_active

This function reports whether a region with all the hints in `hints` is active for the caller. On return `active` is 1 if so and 0 otherwise. It is callable from host or device code.

```
integer(4) function nvshmemx_region_is_active(hints, active)
    integer(4), value :: hints
    integer(4) :: active
```

Chapter 17. NVTX Profiling Library APIs

This chapter describes the Fortran interfaces to the NVIDIA Tools Extension (NVTX) library. NVTX is a set of functions that a developer can use to provide additional information to tools, such as NVIDIA's Nsight Systems performance analysis tool. NVTX functions are accessible from host code, but can be useful in marking and viewing time spans (ranges) of both host and device sections of an application.

The NVTX interfaces and definitions described in this chapter can be exposed by adding the line

```
use nvtx
```

to your program unit. A version of this module has been available through other means in the past, but this chapter documents the Fortran module now included in the NVIDIA HPC SDK. Since we are targeting the NVTX v3 API, a header-only C library, we have instantiated Fortran-callable wrappers and provide those in a library, `libnvhpcwrapnvtx.[a|so]`; linking requires the developer add `-cudalib=nvtx` to their link line, or explicitly add some form of `-lnvhpcwrapnvtx`.

This chapter is divided into three sections. The first describes the traditional Fortran NVTX interfaces which have been available previously. The second describes advanced functions which are now supported in the NVTX v3 API. The third shows a method which leverages the `nvfortran -Minstrument` option to automatically insert NVTX ranges across subprogram entry and exit.

Unless a specific kind is provided, the plain integer type used in the interfaces implies integer(4). Character arguments are accepted as Fortran character strings. Interfaces that also accept `type(nvtxWideString)` pass Unicode strings to the corresponding NVTX wide-character API.

17.1. NVTX Basic Tooling APIs

This section describes the most basic Fortran interfaces to the NVIDIA Tools Extension (NVTX) library. These interfaces were first defined in blog posts and via a publicly available source repository. The simplest interfaces merely push and pop user-labeled, nested time ranges.

The StartRange/EndRange names were transposed from the advanced RangeStart/RangeEnd originally for ease-of-use. Both types can be used in the same program.

17.1.1. nvtxStartRange

This subroutine begins a simple labelled time span range using the NVTX library. The `icolor` argument is optional, and will map to one of many predefined colors. The ranges can be nested. If the optional `id` argument is present, `nvtxStartRange` starts a process range and returns a range ID; otherwise it starts a nested thread range.

```
subroutine nvtxStartRange( message, icolor, id )
  character(len=*) :: message
  integer(4), optional :: icolor
  integer(8), optional :: id
```

17.1.2. nvtxEndRange

This subroutine terminates a simple labelled time span range initiated by `nvtxStartRange`. It takes no arguments when ending a nested thread range. If `nvtxStartRange` was called with the optional `id` argument, pass that ID to `nvtxEndRange` to end the process range.

```
subroutine nvtxEndRange()

subroutine nvtxEndRange(id)
  integer(8) :: id
```

17.2. NVTX Advanced Tooling APIs

This section describes the advanced Fortran interfaces to the NVIDIA Tools Extension (NVTX) library which target the NVTX v3 API.

17.2.1. NVTX Definitions and Derived Types

This section contains the definitions and data types used in the advanced Fortran interfaces to the NVIDIA Tools Extension (NVTX) library, v3 API.

```
integer, parameter :: NVTX_VERSION = 3
integer, parameter :: NVTX_EVENT_ATTRIB_STRUCT_SIZE = 48
integer, parameter :: NVTX_RESOURCE_ATTRIB_STRUCT_SIZE = 32
integer, parameter :: NVTX_NO_PUSH_POP_TRACKING = -2
```

```
integer, parameter :: NVTX_RESOURCE_CLASS_GENERIC = 1
integer, parameter :: NVTX_RESOURCE_TYPE_UNKNOWN = 0
integer, parameter :: NVTX_RESOURCE_TYPE_GENERIC_POINTER = &
    ishft(NVTX_RESOURCE_CLASS_GENERIC, 16) + 1
integer, parameter :: NVTX_RESOURCE_TYPE_GENERIC_HANDLE = &
    ishft(NVTX_RESOURCE_CLASS_GENERIC, 16) + 2
integer, parameter :: NVTX_RESOURCE_TYPE_GENERIC_THREAD_NATIVE = &
    ishft(NVTX_RESOURCE_CLASS_GENERIC, 16) + 3
```

(continues on next page)

(continued from previous page)

```
integer, parameter :: NVTX_RESOURCE_TYPE_GENERIC_THREAD_POSIX = &
    ishft(NVTX_RESOURCE_CLASS_GENERIC, 16) + 4
```

```
enum, bind(C)
    enumerator :: NVTX_SUCCESS = 0
    enumerator :: NVTX_FAIL = 1
    enumerator :: NVTX_ERR_INIT_LOAD_PROPERTY = 2
    enumerator :: NVTX_ERR_INIT_ACCESS_LIBRARY = 3
    enumerator :: NVTX_ERR_INIT_LOAD_LIBRARY = 4
    enumerator :: NVTX_ERR_INIT_MISSING_LIBRARY_ENTRY_POINT = 5
    enumerator :: NVTX_ERR_INIT_FAILED_LIBRARY_ENTRY_POINT = 6
    enumerator :: NVTX_ERR_NO_INJECTION_LIBRARY_AVAILABLE = 7
end enum
```

```
type, bind(c) :: nvtxColorType
  integer(4) :: type
end type
type(nvtxColorType), parameter :: &
    NVTX_COLOR_UNKNOWN  = nvtxColorType(0), &
    NVTX_COLOR_ARGB     = nvtxColorType(1)
```

```
type, bind(c) :: nvtxMessageType
  integer(4) :: type
end type
type(nvtxMessageType), parameter :: &
  NVTX_MESSAGE_UNKNOWN          = nvtxMessageType(0), &
  NVTX_MESSAGE_TYPE_ASCII       = nvtxMessageType(1), &
  NVTX_MESSAGE_TYPE_UNICODE     = nvtxMessageType(2), &
  NVTX_MESSAGE_TYPE_REGISTERED  = nvtxMessageType(3)
```

```
type, bind(c) :: nvtxPayloadType
  integer(4) :: type
end type
type(nvtxPayloadType), parameter :: &
  NVTX_PAYLOAD_UNKNOWN             = nvtxPayloadType(0), &
  NVTX_PAYLOAD_TYPE_UNSIGNED_INT64 = nvtxPayloadType(1), &
  NVTX_PAYLOAD_TYPE_INT64          = nvtxPayloadType(2), &
  NVTX_PAYLOAD_TYPE_DOUBLE         = nvtxPayloadType(3), &
  NVTX_PAYLOAD_TYPE_UNSIGNED_INT32 = nvtxPayloadType(4), &
  NVTX_PAYLOAD_TYPE_INT32          = nvtxPayloadType(5), &
  NVTX_PAYLOAD_TYPE_FLOAT          = nvtxPayloadType(6)
```

```
type nvtxFtnStringType
  character(1), allocatable :: chars(:)
end type
```

```
type nvtxWideString
  integer(c_int32_t), allocatable :: code_units(:)
end type
```

```
type :: nvtxEventAttributes
  integer(C_INT16_T)      :: version = NVTX_VERSION
  integer(C_INT16_T)      :: size = NVTX_EVENT_ATTRIB_STRUCT_SIZE
  integer(C_INT)          :: category = 0
```

(continues on next page)

(continued from previous page)

```
  type(nvtxColorType)     :: colorType = NVTX_COLOR_ARGB
  integer(C_INT)          :: color = -1
  type(nvtxPayloadType)   :: payloadType = NVTX_PAYLOAD_UNKNOWN
  integer(C_INT)          :: reserved0
  integer(C_INT64_T)      :: payload
  type(nvtxMessageType)   :: messageType = NVTX_MESSAGE_TYPE_ASCII
  type(nvtxFtnStringType) :: message
end type
```

```
type :: nvtxResourceAttributes
  integer(C_INT16_T)      :: version = NVTX_VERSION
  integer(C_INT16_T)      :: size = NVTX_RESOURCE_ATTRIB_STRUCT_SIZE
  integer(C_INT)          :: identifierType = NVTX_RESOURCE_TYPE_UNKNOWN
  integer(C_INT64_T)      :: identifier = 0
  type(nvtxMessageType)   :: messageType = NVTX_MESSAGE_UNKNOWN
  type(nvtxFtnStringType) :: message
end type
```

```
type nvtxRangeId
  integer(8) :: id
end type
```

```
type nvtxDomainHandle
  type(C_PTR) :: handle
end type
```

```
type nvtxStringHandle
  type(C_PTR) :: handle
end type
```

```
type nvtxResourceHandle
  type(C_PTR) :: handle
end type
```

17.2.2. nvtxInitialize

This subroutine forces the NVTX library to initialize. It can be used to move the initialization overhead for timing purposes. It takes no arguments.

```
subroutine nvtxInitialize()
```

17.2.3. nvtxDomainCreate

This function creates a new named NVTX domain. Each domain maintains its own push and pop stack.

```
function nvtxDomainCreate(message) result(domain)
   character(len=*) :: message
   type(nvtxDomainHandle) :: domain

function nvtxDomainCreate(message) result(domain)
   type(nvtxWideString) :: message
   type(nvtxDomainHandle) :: domain
```

17.2.4. nvtxDomainDestroy

This subroutine destroys an NVTX domain.

```
subroutine nvtxDomainDestroy(domain)
   type(nvtxDomainHandle) :: domain
```

17.2.5. nvtxDomainRegisterString

This function registers an immutable string with NVTX, for use with the `type(nvtxEventAttributes)` message field.

```
function nvtxDomainRegisterString(domain, message) &
  result(stringHandle)
  type(nvtxDomainHandle) :: domain
  character(len=*) :: message
  type(nvtxStringHandle) :: stringHandle

function nvtxDomainRegisterString(domain, message) &
  result(stringHandle)
  type(nvtxDomainHandle) :: domain
  type(nvtxWideString) :: message
  type(nvtxStringHandle) :: stringHandle
```

Using overloaded assignment defined in this module, users can enable a registered string using these two statements:

```
event%message = nvtxDomainRegisterString(domain, "Str 1")
event%messageType = NVTX_MESSAGE_TYPE_REGISTERED
```

A `type(nvtxEventAttributes)` variable can also be initialized by passing a registered string to the type constructor, along with an optional color:

```
regstr = nvtxDomainRegisterString(domain, "Str 2")
  event = nvtxEventAttributes(regstr, icolor)
```

17.2.6. nvtxDomainNameCategory

This subroutine allows the user to assign a name to a category ID that is specific to the domain.

```
subroutine nvtxDomainNameCategory(domain, category, name)
  type(nvtxDomainHandle) :: domain
  integer(4) :: category
  character(len=*) :: name

subroutine nvtxDomainNameCategory(domain, category, name)
  type(nvtxDomainHandle) :: domain
  integer(4) :: category
  type(nvtxWideString) :: name
```

17.2.7. nvtxNameCategory

This subroutine allows the user to assign a name to a category ID.

```
subroutine nvtxNameCategory(category, name)
  integer(4) :: category
  character(len=*) :: name

subroutine nvtxNameCategory(category, name)
  integer(4) :: category
  type(nvtxWideString) :: name
```

17.2.8. nvtxNameOsThread

This subroutine assigns a name to an operating system thread ID.

```
subroutine nvtxNameOsThread(threadId, name)
  integer(4) :: threadId
  character(len=*) :: name

subroutine nvtxNameOsThread(threadId, name)
  integer(4) :: threadId
  type(nvtxWideString) :: name
```

17.2.9. nvtxDomainResourceCreate

This function creates an NVTX resource object within a domain, using user-supplied NVTX resource attributes.

```
function nvtxDomainResourceCreate(domain, attribs) result(resource)
  type(nvtxDomainHandle) :: domain
  type(nvtxResourceAttributes) :: attribs
  type(nvtxResourceHandle) :: resource
```

17.2.10. nvtxDomainResourceDestroy

This subroutine destroys an NVTX resource object.

```
subroutine nvtxDomainResourceDestroy(resource)
  type(nvtxResourceHandle) :: resource
```

17.2.11. nvtxDomainMarkEx

This subroutine marks an instantaneous event in the application, with full control over the NVTX domain and event attributes.

```
subroutine nvtxDomainMarkEx(domain, event)
  type(nvtxDomainHandle) :: domain
  type(nvtxEventAttributes) :: event
```

17.2.12. nvtxMarkEx

This subroutine marks an instantaneous event in the application, with user-supplied NVTX event attributes.

```
subroutine nvtxMarkEx(event)
  type(nvtxEventAttributes) :: event
```

17.2.13. nvtxMark

This subroutine marks an instantaneous event in the application with a user-supplied message.

```
subroutine nvtxMark(message)
  character(len=*) :: message

subroutine nvtxMark(message)
  type(nvtxWideString) :: message
```

17.2.14. nvtxDomainRangeStartEx

This function starts a process range in the application, with full control over the NVTX domain and event attributes, and returns a unique range ID.

```
function nvtxDomainRangeStartEx(domain, event) result(id)
  type(nvtxDomainHandle) :: domain
  type(nvtxEventAttributes) :: event
  type(nvtxRangeId) :: id
```

17.2.15. nvtxRangeStartEx

This function starts a process range in the application, with user-supplied NVTX event attributes, and returns a unique range ID.

```
function nvtxRangeStartEx(event) result(id)
  type(nvtxEventAttributes) :: event
  type(nvtxRangeId) :: id
```

17.2.16. nvtxRangeStart

This function starts a process range in the application with a user-supplied message, and returns a unique range ID.

```
function nvtxRangeStart(message) result(id)
  character(len=*) :: message
  type(nvtxRangeId) :: id

function nvtxRangeStart(message) result(id)
  type(nvtxWideString) :: message
  type(nvtxRangeId) :: id
```

17.2.17. nvtxDomainRangeEnd

This subroutine ends a process range in the application. Arguments are the domain and range ID from a previous call to `nvtxDomainRangeStartEx`.

```
subroutine nvtxDomainRangeEnd(domain, id)
  type(nvtxDomainHandle) :: domain
  type(nvtxRangeId) :: id
```

17.2.18. nvtxRangeEnd

This subroutine ends a process range in the application. The argument is a range ID returned from a previous call to any nvtxRangeStart function.

```
subroutine nvtxRangeEnd(id)
  type(nvtxRangeId) :: id
```

17.2.19. nvtxDomainRangePushEx

This function starts a nested thread range in the application, with full control over the NVTX domain and event attributes, and returns nested range level.

```
function nvtxDomainRangePushEx(domain, event) result(ilvl)
  type(nvtxDomainHandle) :: domain
  type(nvtxEventAttributes) :: event
  integer(4) :: ilvl
```

17.2.20. nvtxRangePushEx

This function starts a nested thread range in the application, with user-supplied event attributes, and returns the nested range level.

```
function nvtxRangePushEx(event) result(ilvl)
  type(nvtxEventAttributes) :: event
  integer(4) :: ilvl
```

17.2.21. nvtxRangePush

This function starts a nested range in the application with a user-supplied message, and returns the level of the range being started.

```
function nvtxRangePush(message) result(ilvl)
  character(len=*) :: message
  integer(4) :: ilvl

function nvtxRangePush(message) result(ilvl)
  type(nvtxWideString) :: message
  integer(4) :: ilvl
```

17.2.22. nvtxDomainRangePop

This function ends a nested thread range in the application, within a specific domain.

```
function nvtxDomainRangePop(domain) result(ilvl)
  type(nvtxDomainHandle) :: domain
  integer(4) :: ilvl
```

17.2.23. nvtxRangePop

This function ends a nested thread range in the application, and returns the level of the range being ended.

```
function nvtxRangePop() result(ilvl)
  integer(4) :: ilvl
```

17.3. NVTX Automated Instrumentation

This section describes a method to automatically insert NVIDIA Tools Extension (NVTX) ranges into your code without making source changes. This method is only supported on Linux systems.

The first step is to determine which source files you want to view NVTX labels for. In your build process, add this compiler option for those files:

```
-Minstrument
```

This standard compiler option instructs the compiler to insert two calls into the generated code: at subprogram entry, it will insert a call to `__cyg_profile_func_enter()`, and at subprogram exit, it will insert a call to `__cyg_profile_func_exit()`. These entry points are meant to be supplied by profiling tools. One important input argument to these functions, inserted by the compiler, is the function address.

The next step, for best user experience, is to link your executable with these options:

```
-traceback  -lnvhpcwrapnvtx
```

or alternatively:

```
-fPIC  -Wl,-export-dynamic  -lnvhpcwrapnvtx
```

These options will enable the runtime to convert the function or subroutine address into a symbol, via the dladdr() system call. Without these options, the label will contain the subprogram unit address, in hexadecimal, which is useful, but does require some other manual processing steps to determine the associated symbol name.

As with all of the NVTX instrumentation methods, you need to enable the processing of the NVTX API calls when you run. An example of enabling NVTX, using Nsight Systems, is to use

```
nsys profile --trace=nvtx
```

which will result in the NVTX time span ranges presented on the Nsight timeline. Currently,

```
--trace=nvtx
```

is set by default, so just specifying

```
nsys profile ./a.out
```

will provide you with the NVTX annotations, along with CUDA traces.

Chapter 18. NVIDIA Communication Abstraction Library (CAL) APIs

This section describes the Fortran interfaces to the CAL library. This library contains a small number of functions used in conjunction with other multi-processor libraries, for initializing a communications layer used for handling distributed matrices and arrays.

The CAL interfaces and definitions described in this chapter can be exposed in host code by adding the line

```
use nvf_cal_comm
```

to your program unit, but since the module definitions are also used within other modules in this document, it is suggested you access them through the higher-level modules, such as

```
use cublasMp
```

or

```
use cusolverMp
```

.

18.1. CAL API

This section describes the parameters and derived types defined in the CAL module.

```
! Definitions from cal.h
integer, parameter :: CAL_VER_MAJOR = 0
integer, parameter :: CAL_VER_MINOR = 4
integer, parameter :: CAL_VER_PATCH = 2
integer, parameter :: CAL_VER_BUILD = 0
integer, parameter :: CAL_VERSION = &
        (CAL_VER_MAJOR * 1000 + CAL_VER_MINOR * 100 + CAL_VER_PATCH)
```

```
enum, bind(c)
    enumerator :: CAL_OK = 0                        ! Success
    enumerator :: CAL_ERROR_INPROGRESS = 1          ! Request is in progress
```

(continues on next page)

(continued from previous page)

```
    enumerator :: CAL_ERROR = 2                    ! Generic error
    enumerator :: CAL_ERROR_INVALID_PARAMETER = 3  ! Invalid parameter to the
↪interface function.
    enumerator :: CAL_ERROR_INTERNAL = 4           ! Internal error
    enumerator :: CAL_ERROR_CUDA = 5               ! Error in CUDA runtime/driver API
    enumerator :: CAL_ERROR_UCC = 6                ! Error in UCC call
    enumerator :: CAL_ERROR_NOT_SUPPORTED = 7      ! Requested configuration not
↪supported
end enum
```

```
! Types from cal.h
    TYPE cal_comm
      TYPE(C_PTR) :: handle
    END TYPE cal_comm
```

18.1.1. cal_comm_create_mpi

CAL is a communications library used by cublasMp and other multi-processor libraries. It uses MPI for initialization and potentially other uses. This is a convenience function provided with Fortran to initialize the cal communicator. Because of the dependence on MPI, source code for some cal Fortran wrappers are shipped in the NVHPC package, and can be built with the MPI headers used in your application. The version we ship works with the default MPI bundled in the NVHPC package. The cal communicator derived type output by this function is an input to `cublasMpGridCreate()`, `cusolverMpCreateDeviceGrid()` and similar functions.

```
integer(4) function cal_comm_create_mpi(mpi_comm, &
      rank, nranks, local_device, comm)
  integer(4), intent(in) :: mpi_comm, rank, nranks, local_device
  type(cal_comm), intent(out) :: comm
```

18.1.2. cal_comm_destroy

This function destroys the cal_comm data structure and frees the resources associated with it.

```
integer(4) function cal_comm_destroy(comm)
  type(cal_comm) :: comm
```

18.1.3. cal_stream_sync

This function blocks the calling thread until all outstanding device operations, including cal operations, are finished in the specified `stream`.

```
integer(4) function cal_stream_sync(comm, stream)
  type(cal_comm) :: comm
  integer(cuda_stream_kind) :: stream
```

18.1.4. cal_comm_barrier

This function synchronizes streams from all processes in the cal communicator, basically an all-to-all synchronization.

```
integer(4) function cal_comm_barrier(comm, stream)
  type(cal_comm) :: comm
  integer(cuda_stream_kind) :: stream
```

18.1.5. cal_comm_get_rank

This function returns the rank of the calling thread in the CAL communicator.

```
integer(4) function cal_comm_get_rank(comm, rank)
  type(cal_comm) :: comm
  integer(4) :: rank
```

18.1.6. cal_comm_get_size

This function returns the size of the CAL communicator.

```
integer(4) function cal_comm_get_size(comm, size)
  type(cal_comm) :: comm
  integer(4) :: size
```

Chapter 19. Examples

This section contains examples with source code.

19.1. Using cuBLAS from OpenACC Host Code

This example demonstrates the use of the cublas module, the cublasHandle type, and several forms of blas calls from OpenACC data regions.

Simple OpenACC BLAS Test

```
program testcublas
! compile with pgfortran -ta=tesla -cudalib=cublas -cuda testcublas.f90
call testcu1(1000)
call testcu2(1000)
end
!
subroutine testcu1(n)
use openacc
use cublas
integer :: a(n), b(n)
type(cublasHandle) :: h
istat = cublasCreate(h)
! Force OpenACC kernels and cuBLAS to use the OpenACC stream.
istat = cublasSetStream(h, acc_get_cuda_stream(acc_async_sync))
!$acc data copyout(a, b)
!$acc kernels
a = 1
b = 2
!$acc end kernels
! No host_data, we are lexically inside a data region
! sswap will accept any kind(4) data type
call sswap(n, a, 1, b, 1)
call cublasSswap(n, a, 1, b, 1)
!$acc end data
if (all(a.eq.1).and.all(b.eq.2)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testcu2(n)
```

(continues on next page)

(continued from previous page)

```
use openacc
use cublas
real(8) :: a(n), b(n)
a = 1.0d0
b = 2.0d0
!$acc data copy(a, b)
!$acc host_data use_device(a,b)
call dswap(n, a, 1, b, 1)
call cublasDswap(n, a, 1, b, 1)
!$acc end host_data
!$acc end data
if (all(a.eq.1.0d0).and.all(b.eq.2.0d0)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
```

CUBLASXT BLAS Test

This example demonstrates the use of the cublasxt module

```
program testcublasxt
call testxt1(1000)
call testxt2(1000)
end
!
subroutine testxt1(n)
use cublasxt
real(4) :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = 1.0
b = 2.0
c = -1.0
alpha = 1.0
beta = 0.0
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtSgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(c.eq.2.0*n)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
print *,c(1,1),c(n,n)
end
!
subroutine testxt2(n)
```

(continues on next page)

(continued from previous page)

```
use cublasxt
real(8) :: a(n,n), b(n,n), c(n,n), alpha, beta
type(cublasXtHandle) :: h
integer ndevices(1)
a = 1.0d0
b = 2.0d0
c = -1.0d0
alpha = 1.0d0
beta = 0.0
istat = cublasXtCreate(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
ndevices(1) = 0
istat = cublasXtDeviceSelect(h, 1, ndevices)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDgemm(h, CUBLAS_OP_N, CUBLAS_OP_N, &
                      n, n, n, &
                      alpha, A, n, B, n, beta, C, n)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
istat = cublasXtDestroy(h)
if (istat .ne. CUBLAS_STATUS_SUCCESS) print *,istat
if (all(c.eq.2.0d0*n)) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
print *,c(1,1),c(n,n)
end
```

19.2. Using cuBLAS from CUDA Fortran Host Code

This example demonstrates the use of the cublas module, the cublasHandle type, and several forms of blas calls.

Simple BLAS Test

```
program testisamax
! Compile with "pgfortran testisamax.cuf -cudalib=cublas -lblas"
! Use the NVIDIA cudafor and cublas modules
use cudafor
use cublas
!
real*4, device, allocatable :: xd(:)
real*4 x(1000)
integer, device :: kd
type(cublasHandle) :: h

call random_number(x)

! F90 way
i = maxloc(x,dim=1)
print *,i
```

(continues on next page)

(continued from previous page)

```
print *,x(i-1),x(i),x(i+1)

! Host way
j = isamax(1000,x,1)
print *,j
print *,x(j-1),x(j),x(j+1)

! CUDA Generic BLAS way
allocate(xd(1000))
xd = x
k = isamax(1000,xd,1)
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA Specific BLAS way
k = cublasIsamax(1000,xd,1)
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA V2 Host Specific BLAS way
istat = cublasCreate(h)
if (istat .ne. 0) print *,"cublasCreate returned ",istat
k = 0
istat = cublasIsamax_v2(h, 1000, xd, 1, k)
if (istat .ne. 0) print *,"cublasIsamax 1 returned ",istat
print *,k
print *,x(k-1),x(k),x(k+1)

! CUDA V2 Device Specific BLAS way
k = 0
istat = cublasIsamax_v2(h, 1000, xd, 1, kd)
if (istat .ne. 0) print *,"cublasIsamax 2 returned ",istat
k = kd
print *,k
print *,x(k-1),x(k),x(k+1)

istat = cublasDestroy(h)
if (istat .ne. 0) print *,"cublasDestroy returned ",istat

end program
```

Multi-threaded BLAS Test

This example demonstrates the use of the cublas module in a multi-threaded code. Each thread will attach to a different GPU, create a context, and combine the results at the end.

```
program tsgemm
!
! Multi-threaded example calling sgemm from cuda fortran.
! Compile with "pgfortran -mp tsgemm.cuf -cudalib=cublas"
! Set OMP_NUM_THREADS=number of GPUs in your system.
!
use cublas
use cudafor
use omp_lib
!
! Size this according to number of GPUs
```

(continues on next page)

(continued from previous page)

```
!
! Small
!integer, parameter :: K = 2500
!integer, parameter :: M = 2000
!integer, parameter :: N = 2000
! Large
integer, parameter :: K = 10000
integer, parameter :: M = 10000
integer, parameter :: N = 10000
integer, parameter :: NTIMES = 10
!
real*4, device, allocatable :: a_d(:,:), b_d(:,:), c_d(:,:)
!$omp THREADPRIVATE(a_d,b_d,c_d)
real*4 a(m,k), b(k,n), c(m,n)
real*4 alpha, beta
integer, allocatable :: offs(:)
type(cudaEvent) :: start, stop

a = 1.0; b = 0.0; c = 0.0
do i = 1, N
  b(i,i) = 1.0
end do
alpha = 1.0; beta  = 1.0

! Break up the B and C array into sections
nthr = omp_get_max_threads()
nsec = N / nthr
print *,"Running with ",nthr," threads, each section = ",nsec
allocate(offs(nthr))
offs = (/ (i*nsec,i=0,nthr-1) /)

! Allocate and initialize the arrays
! Each thread connects to a device and creates a CUDA context.
!$omp PARALLEL private(i,istat)
    i = omp_get_thread_num() + 1
    istat = cudaSetDevice(i-1)
    allocate(a_d(M,K), b_d(K,nsec), c_d(M,nsec))
    a_d = a
    b_d = b(:,offs(i)+1:offs(i)+nsec)
    c_d = c(:,offs(i)+1:offs(i)+nsec)
!$omp end parallel

istat = cudaEventCreate(start)
istat = cudaEventCreate(stop)

time = 0.0
istat = cudaEventRecord(start, 0)
! Run the traditional blas kernel
!$omp PARALLEL private(j,istat)
    do j = 1, NTIMES
      call sgemm('n','n', M, N/nthr, K, alpha, a_d, M, b_d, K, beta, c_d, M)
    end do
    istat = cudaDeviceSynchronize()
!$omp end parallel

istat = cudaEventRecord(stop, 0)
```

(continues on next page)

(continued from previous page)

```
istat = cudaEventElapsedTime(time, start, stop)
time = time / (NTIMES*1.0e3)
!$omp PARALLEL private(i)
    i = omp_get_thread_num() + 1
    c(:,offs(i)+1:offs(i)+nsec) = c_d
!$omp end parallel
nerrors = 0
do j = 1, N
    do i = 1, M
        if (c(i,j) .ne. NTIMES) nerrors = nerrors + 1
    end do
end do
print *,"Number of Errors:",nerrors
gflops = 2.0 * N * M * K/time/1e9
write (*,901) m,k,k,N,time*1.0e3,gflops
901 format(i0,'x',i0,' * ',i0,'x',i0,':\t',f8.3,' ms\t',f12.3,' GFlops/s')
end
```

19.3. Using cuFFT from OpenACC Host Code

This example demonstrates the use of the cufft module, the cufftHandle type, and several cuFFT library calls.

Simple cuFFT Test

```
program cufft2dTest
  use cufft
  use openacc
  integer, parameter :: m=768, n=512
  complex, allocatable  :: a(:,:),b(:,:),c(:,:)
  real, allocatable     :: r(:,:),q(:,:)
  integer :: iplan1, iplan2, iplan3, ierr

  allocate(a(m,n),b(m,n),c(m,n))
  allocate(r(m,n),q(m,n))

  a = 1; r = 1
  xmx = -99.0

  ierr = cufftPlan2D(iplan1,n,m,CUFFT_C2C)
  ierr = ierr + cufftSetStream(iplan1,acc_get_cuda_stream(acc_async_sync))
  !$acc host_data use_device(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$acc end host_data

  ! scale c
  !$acc kernels
  c = c / (m*n)
  !$acc end kernels

  ! Check forward answer
  write(*,*) 'Max error C2C FWD: ', cmplx(maxval(real(b)) - sum(real(b)), &
```

(continues on next page)

(continued from previous page)

```
                                          maxval(imag(b)))
  ! Check inverse answer
  write(*,*) 'Max error C2C INV: ', maxval(abs(a-c))

  ! Real transform
  ierr = ierr + cufftPlan2D(iplan2,n,m,CUFFT_R2C)
  ierr = ierr + cufftPlan2D(iplan3,n,m,CUFFT_C2R)
  ierr = ierr + cufftSetStream(iplan2,acc_get_cuda_stream(acc_async_sync))
  ierr = ierr + cufftSetStream(iplan3,acc_get_cuda_stream(acc_async_sync))

  !$acc host_data use_device(r,b,q)
  ierr = ierr + cufftExecR2C(iplan2,r,b)
  ierr = ierr + cufftExecC2R(iplan3,b,q)
  !$acc end host_data

  !$acc kernels
  xmx = maxval(abs(r-q/(m*n)))
  !$acc end kernels

  ! Check R2C + C2R answer
  write(*,*) 'Max error R2C/C2R: ', xmx

  ierr = ierr + cufftDestroy(iplan1)
  ierr = ierr + cufftDestroy(iplan2)
  ierr = ierr + cufftDestroy(iplan3)

  if (ierr.eq.0) then
    print *,"test PASSED"
  else
    print *,"test FAILED"
  endif

end program cufft2dTest
```

19.4. Using cuFFT from CUDA Fortran Host Code

This example demonstrates the use of the cuFFT module, the cufftHandle type, and several cuFFT library calls.

Simple cuFFT Test

```
program cufft2dTest
  use cudafor
  use cufft
  implicit none
  integer, parameter :: m=768, n=512
  complex, managed :: a(m,n),b(m,n)
  real, managed :: ar(m,n),br(m,n)
  real    x
  integer plan, ierr
  logical passing
```

(continues on next page)

(continued from previous page)

```
  a = 1; ar = 1

  ierr = cufftPlan2D(plan,n,m,CUFFT_C2C)
  ierr = ierr + cufftExecC2C(plan,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(plan,b,b,CUFFT_INVERSE)
  ierr = ierr + cudaDeviceSynchronize()
  x = maxval(abs(a-b/(m*n)))
  write(*,*) 'Max error C2C: ', x
  passing = x .le. 1.0e-5

  ierr = ierr + cufftPlan2D(plan,n,m,CUFFT_R2C)
  ierr = ierr + cufftExecR2C(plan,ar,b)
  ierr = ierr + cufftPlan2D(plan,n,m,CUFFT_C2R)
  ierr = ierr + cufftExecC2R(plan,b,br)
  ierr = ierr + cudaDeviceSynchronize()
  x = maxval(abs(ar-br/(m*n)))
  write(*,*) 'Max error R2C/C2R: ', x
  passing = passing .and. (x .le. 1.0e-5)

  ierr = ierr + cufftDestroy(plan)
  print *,ierr
  passing = passing .and. (ierr .eq. 0)
  if (passing) then
    print *,"Test PASSED"
  else
    print *,"Test FAILED"
  endif
end program cufft2dTest
```

19.5. Using cufftXt from CUDA Fortran Host Code

This example demonstrates the use of the cufftXt module, the cudaLibXtDesc type, many of the cufftXt API calls, the padding required for real/complex FFTs, and the distribution of active modes across multiple GPUs.

Simple Poisson 3D CufftXt Test

```
program cufftXt3DTest
  use cudafor
  use cufftXt   ! cufftXt module includes cufft
  implicit none

  integer, parameter :: nGPUs = 2
  integer, parameter :: M = 1, N = 1, P = 1
  integer, parameter :: nx = 32, ny=32, nz=32
  real, parameter :: twopi = 8.0*atan(1.0)
  real, parameter :: Lx = 1.0, Ly = 1.0, Lz = 1.0
  logical, parameter :: printModes = .true.

  ! GPU
```

(continues on next page)

(continued from previous page)

```
integer :: nodeGPUs
integer :: whichGPUs(nGPUs)

! grid
real :: x(nx), y(ny), z(nz)
real :: hx, hy, hz

! wavenumbers
real :: kx(nx/2+1), ky(ny), kz(nz)
type realDeviceArray
   real, device, allocatable :: v(:)
end type realDeviceArray
type(realDeviceArray) :: kx_da(nGPUs), ky_da(nGPUs), kz_da(nGPUs)

real :: phi(nx+2,ny, nz), u(nx+2,ny,nz), ua(nx,ny,nz)
integer :: status, i, j, k

! check M, N

if (M==0 .or. N==0 .or. P==0) then
   write(*,*) 'M, N or P is 0 thus solution is u=0'
   stop
else
   write(*,"(' Running with modes M, N, P = ', i0, ', ', i0, ', ', i0)") M, N, P
   write(*,"('  on a grid of ', i0, 'x', i0, 'x', i0)") nx, ny, nz
end if

! choose GPUs

! check for multiple GPUs
status = cudaGetDeviceCount(nodeGPUs)
if (status /= cudaSuccess) write(*,*) 'Error in cudaGetDeviceCount()'
if (nodeGPUs .lt. nGPUs) then
   write(*,*) 'Number of GPUs on node:', nodeGPUs
   write(*,*) 'Insufficient number of GPUs for this code, exiting ...'
   stop
end if

! check for GPUs of same compute capability
block
  type(cudaDeviceProp) :: prop
  integer :: cc(0:nodeGPUs-1,2)
  logical :: foundIdenticalGPUs = .false.
  integer :: iWG, nIdentGPUs

  write(*,*) 'Available devices:'
  do i = 0, nodeGPUs-1
     status = cudaGetDeviceProperties(prop, i)
     if (status /= cudaSuccess) write(*,*) 'Error in cudaGetDeviceProperties()'
     cc(i,1) = prop%major
     cc(i,2) = prop%minor
     write(*,"('   ', i0, ' CC: ', i0, '.', i0)") i, cc(i,1), cc(i,2)
  enddo

  do i = 0, nodeGPUs-1
     nIdentGPUs = 1
```

(continues on next page)

(continued from previous page)

```
      do j = i+1, nodeGPUs
         if (all(cc(i,:) == cc(j,:))) nIdentGPUs = nIdentGPUs+1
      enddo

      if (nIdentGPUs .ge. nGPUs) then
         foundIdenticalGPUs = .true.
         iWG = 1
         whichGPUs(iWG) = i
         do j = i+1, nodeGPUs-1
            if (all(cc(i,:) == cc(j,:))) then
               iWG = iWG+1
               whichGPUs(iWG) = j
            end if
            if (iWG == nGPUs) exit
         end do
         exit
      end if
   end do

   if (foundIdenticalGPUS) then
      write(*,*) 'Running on GPUs:'
      write(*,*) whichGPUs
   else
      write(*,"(' No ', i0, ' identical GPUs found, exiting ...')"), nGPUs
      stop
   end if

 end block

 ! Physical grid

 hx = Lx/nx
 do i = 1, nx
    x(i) = hx*i
 enddo

 hy = Ly/ny
 do j = 1, ny
    y(j) = hy*j
 enddo

 hz = Lz/nz
 do k = 1, nz
    z(k) = hz*k
 enddo

 ! Wavenumbers

 do i = 1, nx/2+1
    kx(i) = (i-1)*(twoPi/Lx)
 enddo

 do j = 1, ny/2
    ky(j) = (j-1)*(twoPi/Ly)
 enddo
 do j = ny/2+1, ny
```

(continues on next page)

(continued from previous page)

```
   ky(j) = (j-1-ny)*(twoPi/Ly)
enddo

do k = 1, nz/2
   kz(k) = (k-1)*(twoPi/Lz)
enddo
do k = nz/2+1, nz
   kz(k) = (k-1-nz)*(twoPi/Lz)
enddo

! copy wavenumber arrays to each device
do i = 1, nGPUs
   status = cudaSetDevice(whichGPUs(i))
   allocate(kx_da(i)%v, source=kx)
   allocate(ky_da(i)%v, source=ky)
   allocate(kz_da(i)%v, source=kz)
enddo

! Initialize phi and get analytical solution

do k = 1, nz
   do j = 1, ny
      do i = 1, nx
         phi(i,j,k) = sin(twoPi*M*x(i))*sin(twoPi*N*y(j))*sin(twoPi*P*z(k))
         ua(i,j,k) = -phi(i,j,k)/((twoPi*M)**2 + (twoPi*N)**2 + (twoPi*P)**2)
      enddo
   enddo
end do

! cufft block

block
  integer :: planR2C, planC2R
  integer(c_size_t) :: worksize(nGPUs)
  type(cudaLibXtDesc), pointer :: phi_desc

  status = cufftCreate(planR2C)
  if (status /= CUFFT_SUCCESS) write(*,*) 'Error in cufftCreate'

  status = cufftCreate(planC2R)
  if (status /= CUFFT_SUCCESS) write(*,*) 'Error in cufftCreate'

  status = cufftXtSetGPUs(planR2C, nGPUs, whichGPUs)
  if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtSetGPUs failed'

  status = cufftXtSetGPUs(planC2R, nGPUs, whichGPUs)
  if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtSetGPUs failed'

  status = cufftMakePlan3d(planR2C, nz, ny, nx, CUFFT_R2C, worksize)
  if (status /= CUFFT_SUCCESS) write(*,*) 'Error in cufftMakePlan2d'

  status = cufftMakePlan3d(planC2R, nz, ny, nx, CUFFT_C2R, worksize)
  if (status /= CUFFT_SUCCESS) write(*,*) 'Error in cufftMakePlan2d'

  ! allocate memory on seperate devices
  status = cufftXtMalloc(planR2C, phi_desc, CUFFT_XT_FORMAT_INPLACE)
```

(continues on next page)

(continued from previous page)

```
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtMalloc failed'

    ! H2D transfer
    write(*,*) 'cufftXtMemcpy H2D ...'
    status = cufftXtMemcpy(planR2C, phi_desc, phi, CUFFT_COPY_HOST_TO_DEVICE)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtMemcpy H2D failed'

    ! forward FFT
    write(*,*) 'Forward transform ...'
    status = cufftXtExecDescriptorR2C(planR2C, phi_desc, phi_desc)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtExecDescriptorR2C failed:', status

    if (printModes) then
       block
         real :: threshold = 1.e-3
         type(cudaXtDesc), pointer :: phiXtDesc
         complex, device, pointer :: phi_d(:,:,:)
         integer :: dev, g, jl, nyl
         complex :: phi_h(nx/2+1, ny, nz)

         call c_f_pointer(phi_desc%descriptor, phiXtDesc)

         do g = 1, nGPUs
            write(*,"(' Active modes on g = ', i0)") g
            dev = phiXtDesc%GPUs(g)
            status = cudaSetDevice(dev)
            if (status /= cudaSuccess) write(*,*) 'cudaSetDevice failed'

            ! XtMalloc done with FORMAT_INPLACE, so data prior to transform
            ! are in natural order (split in least frequently changing index,
            ! in this case z), and after transform are in shuffled order (split in
            ! second least frequently changing index, in this case y)

            nyl = ny/nGPUs
            if (g .le. mod(ny, nGPUs)) nyl = nyl+1

            call c_f_pointer(phiXtDesc%data(g), phi_d, [nx/2+1, nyl, nz])

            !$cuf kernel do (3)
            do k = 1, nz
               do jl = 1, nyl
                  do i = 1, nx/2+1
                     if (abs(phi_d(i,jl,k)) .gt. threshold) print*, i, jl, k, phi_d(i,
↪jl,k)
                  enddo
               enddo
            enddo
            status = cudaDeviceSynchronize()
            call flush()
         end do

         status = cufftXtMemcpy(planR2C, phi_h, phi_desc, CUFFT_COPY_DEVICE_TO_HOST)
         if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtMemcpy D2H failed'
         write(*,*) 'Active modes on host (after D2H):'
         do k = 1, nz
            do j = 1, ny
```

(continues on next page)

(continued from previous page)

```
            do i = 1, nx/2+1
               if (abs(phi_h(i,j,k)) .gt. threshold) print*, i, j, k, phi_h(i,j,k)
            end do
         end do
      enddo
   end block
end if

! solve Poisson equation
write(*,*) 'Solve Poisson Eq ...'
block
  type(cudaXtDesc), pointer :: phiXtDesc
  complex, device, pointer :: phi_d(:,:,:)
  integer :: dev, g, jl, nyl, yOffset
  real :: k2

  call c_f_pointer(phi_desc%descriptor, phiXtDesc)

  yOffset = 0
  do g = 1, nGPUs
     dev = phiXtDesc%GPUs(g)
     status = cudaSetDevice(dev)
     if (status /= cudaSuccess) write(*,*) 'cudaSetDevice failed'

     ! XtMalloc done with FORMAT_INPLACE, so data prior to transform
     ! are in natural order (split in least frequently changing index,
     ! in this case z), and after transform are in shuffled order (split in
     ! second least frequently changing index, in this case y)

     nyl = ny/nGPUs
     if (g .le. mod(ny, nGPUs)) nyl = nyl+1

     call c_f_pointer(phiXtDesc%data(g), phi_d, [nx/2+1, nyl, nz])

     associate(kx_d => kx_da(g)%v, ky_d => ky_da(g)%v, kz_d => kz_da(g)%v)
       !$cuf kernel do (3)
       do k = 1, nz
          do jl = 1, nyl
             j = jl + yOffset
             do i = 1, nx/2+1
                k2 = kx_d(i)**2 + ky_d(j)**2 + kz_d(k)**2
                phi_d(i,jl,k) = -phi_d(i,jl,k)/k2/(nx*ny*nz)
             enddo
          enddo
       enddo
     end associate

     ! specify mean (corrects division by zero wavenumber above)
     if (g == 1) phi_d(1,1,1) = 0.0

     yOffset = yOffset + nyl
  end do

  do g = 1, nGPUs
     dev = phiXtDesc%GPUs(g)
     status = cudaSetDevice(dev)
```

(continues on next page)

(continued from previous page)

```
         if (status /= cudaSuccess) write(*,*) 'cudaSetDevice failed'
         status = cudaDeviceSynchronize()
         if (status /= cudaSuccess) write(*,*) 'CUF kernel sync error'
       end do

    end block ! poisson

    ! inverse FFT
    write(*,*) 'Inverse transform ...'
    status = cufftXtExecDescriptorC2R(planC2R, phi_desc, phi_desc)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtExecDescriptorC2R failed'

    ! D2H transfer
    write(*,*) 'cufftXtMemcpy D2H ...'
    status = cufftXtMemcpy(planC2R, u, phi_desc, CUFFT_COPY_DEVICE_TO_HOST)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtMemcpy D2H failed'

    ! cufft block cleanup
    status = cufftXtFree(phi_desc)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtFree failed'

    status = cufftDestroy(planR2C)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtFree failed'

    status = cufftDestroy(planC2R)
    if (status /= CUFFT_SUCCESS) write(*,*) 'cufftXtFree failed'

  end block

  write(*,*) 'Max error: ', maxval(abs(u(1:nx,1:ny,1:nz)-ua(1:nx,1:ny,1:nz)))

  ! cleanup

  do i = 1, nGPUs
     status = cudaSetDevice(whichGPUs(i))
     deallocate(kx_da(i)%v, ky_da(i)%v, kz_da(i)%v)
  enddo

  write(*,*) '... finished'

end program cufftXt3DTest
```

19.6. Using cuFFTMp from either OpenACC or CUDA Fortran

This example demonstrates the use of the cuFFTMp API from the cuFFTXt module, the cudaLibXtDesc and cudaXtDesc types, and attaching the MPI COMM to the cuFFT plan.

Real-to-Complex and Complex-to-Real cuFFTMp Test

```
!
! This samples illustrates a basic use of cuFFTMp using the built-in, optimized,
```

(continues on next page)

(continued from previous page)

```
! data distributions.
!
!  It assumes the CPU data is initially distributed according to
!  CUFFT_XT_FORMAT_INPLACE, a.k.a. X-Slabs.
!  Given a global array of size X!  Y!  Z, every MPI rank owns approximately
!  (X / ngpus)!  Y*Z entries.
!  More precisely,
!  - The first (ngpus % X) MPI rank each own (X/ngpus+1) planes of size Y*Z,
!  - The remaining MPI rank each own (X / ngpus) planes of size Y*Z
!
!  The CPU data is then copied on GPU and a forward transform is applied.
!
!  After that transform, GPU data is distributed according to
!  CUFFT_XT_FORMAT_INPLACE_SHUFFLED, a.k.a. Y-Slabs.
!  Given a global array of size X * Y * Z, every MPI rank owns approximately
!  X * (Y / ngpus) * Z entries.
!  More precisely,
!  - The first (ngpus % Y) MPI rank each own (Y/ngpus+1) planes of size X*Z,
!  - The remaining MPI rank each own (Y / ngpus) planes of size X*Z
!
!  A scaling kerel is applied, on the distributed GPU data (distributed
!  according to CUFFT_XT_FORMAT_INPLACE)
!  This kernel prints some elements to illustrate the
!  CUFFT_XT_FORMAT_INPLACE_SHUFFLED data distribution and normalize entries
!  by (nx * ny * nz)
!
!  Finally, a backward transform is applied.
!  After this, data is again distributed according to CUFFT_XT_FORMAT_INPLACE,
!  same as the input data.
!
!  Data is finally copied back to CPU and compared to the input data. They
!  should be almost identical.
!
!  This program can be used by either OpenACC or CUDA Fortran:
!    mpif90 -acc cufftmp_r2c.F90 -cudalib=cufftmp
!  Or:
!    mpif90 -cuda cufftmp_r2c.F90 -cudalib=cufftmp
!
module cufft_required
    integer :: planr2c, planc2r
    integer :: local_rshape(3), local_rshape_permuted(3)
    integer :: local_permuted_cshape(3)
end module cufft_required

program cufftmp_r2c
    use iso_c_binding
    use cufftXt
    use cufft
!@cuf use cudafor
!@acc use openacc
    use mpi
    use cufft_required
    implicit none

    integer :: size, rank, ndevices, ierr
    integer :: nx, ny, nz ! nx slowest
```

(continues on next page)

(continued from previous page)

```
    integer :: i, j, k
    integer :: my_nx, my_ny, my_nz, ranks_cutoff, whichgpu(1)
    real, dimension(:, :, :), allocatable :: u, ref
    complex, dimension(:,:,:), allocatable :: u_permuted
    real :: max_norm, max_diff

    ! cufft stuff
    integer(c_size_t) :: worksize(1)
    type(cudaLibXtDesc), pointer :: u_desc
    type(cudaXtDesc), pointer    :: u_descptr
#ifdef _OPENACC
    complex, pointer             :: u_dptr(:,:,:)
    type(c_ptr)                  :: tmpcptr
#else
    complex, pointer, device     :: u_dptr(:,:,:)
#endif

    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD,size,ierr)
    call mpi_comm_rank(MPI_COMM_WORLD,rank,ierr)

#ifdef _OPENACC
    ndevices = acc_get_num_devices(acc_device_nvidia)
    call acc_set_device_num(mod(rank, ndevices), acc_device_nvidia)
#else
    call checkCuda(cudaGetDeviceCount(ndevices))
    call checkCuda(cudaSetDevice(mod(rank, ndevices)))
#endif
    whichgpu(1) = mod(rank, ndevices)

    print*,"Hello from rank ",rank," gpu id",mod(rank,ndevices),"size",size

    nx = 256
    ny = nx
    nz = nx

    ! We start with X-Slabs
    ! Ranks 0 ... (nx % size - 1) have 1 more element in the X dimension
    ! and every rank own all elements in the Y and Z dimensions.
    ranks_cutoff = mod(nx, size)
    my_nx = nx / size
    if (rank < ranks_cutoff) my_nx = my_nx + 1
    my_ny =  ny;
    my_nz =  nz;
    ! Note nz is first dimension, nx third
    local_rshape = [2*(nz/2+1), ny, my_nx]
    local_permuted_cshape = [nz/2+1, ny/size, nx]
    local_rshape_permuted = [2*(nz/2+1), ny/size, nx]
    if (mod(ny, size) > 0) then
        print*," ny has to divide evenly by mpi_procs"
        call mpi_finalize(ierr)
    end if
    if (rank == 0) then
        write(*,*) "local_rshape          :", local_rshape(1:3)
        write(*,*) "local_permuted_cshape :", local_permuted_cshape(1:3)
    end if
```

(continues on next page)

(continued from previous page)

```
    ! Generate local, distributed data
    allocate(u(local_rshape(1), local_rshape(2), local_rshape(3)))
    allocate(u_permuted(local_permuted_cshape(1), local_permuted_cshape(2), &
        local_permuted_cshape(3)))
    allocate(ref(local_rshape(1), local_rshape(2), local_rshape(3)))
    print*,'shape of u is ', shape(u)
    print*,'shape of u_permuted is ', shape(u_permuted)
    call generate_random(nz, local_rshape(1), local_rshape(2), &
        local_rshape(3), u)
    ref = u
    u_permuted = (0.0,0.0)

    call checkNorm(nz, local_rshape(1), local_rshape(2), local_rshape(3), &
        u, max_norm)
    print*, "initial data on ", rank, " max_norm is ", max_norm

    call checkCufft(cufftCreate(planr2c))
    call checkCufft(cufftCreate(planc2r))
    call checkCufft(cufftMpAttachComm(planr2c, CUFFT_COMM_MPI, &
        MPI_COMM_WORLD), 'cufftMpAttachComm error')
    call checkCufft(cufftMpAttachComm(planc2r, CUFFT_COMM_MPI, &
        MPI_COMM_WORLD), 'cufftMpAttachComm error')

    ! Note nx, ny, nz order
    call checkCufft(cufftMakePlan3d(planr2c, nx, ny, nz, CUFFT_R2C, &
        worksize), 'cufftMakePlan3d r2c error')
    call checkCufft(cufftMakePlan3d(planc2r, nx, ny, nz, CUFFT_C2R, &
        worksize), 'cufftMakePlan3d c2r error')

    call checkCufft(cufftXtMalloc(planr2c, u_desc, &
        CUFFT_XT_FORMAT_INPLACE), 'cufftXtMalloc error')

    ! These are equivalent, and work as well
    ! istat = cufftXtMemcpy(planr2c, u_desc, u, CUFFT_COPY_HOST_TO_DEVICE)
    ! call cufft_memcpyH2D(u_desc, u, CUFFT_XT_FORMAT_INPLACE, .false.)
    call cufft_memcpyH2D(u_desc, u, CUFFT_XT_FORMAT_INPLACE, .true.)

    ! now reset u to make sure the check later is valid
    u = 0.0

    !xxxxxxxxxxxxxxxxxxxxxxxxx Forward
    call checkCufft(cufftXtExecDescriptor(planr2c, u_desc, u_desc, &
        CUFFT_FORWARD),'forward fft failed')

    ! in case we want to check the results after Forward
    !call checkCufft(cufftXtMemcpy(planr2c, u_permuted, u_desc, CUFFT_COPY_DEVICE_TO_
↪HOST), 'permuted D2H error')
    !call checkNormComplex(local_permuted_cshape(1), local_permuted_cshape(2), local_
↪permuted_cshape(3), u_permuted, max_norm)
    !write(*,'(A18, I1, A14, F25.8)') "after R2C ", rank, " max_norm is ", max_norm

    ! Data is now distributed as Y-Slab. We need to scale the output
    call c_f_pointer(u_desc%descriptor, u_descptr)

#ifdef _OPENACC
```

(continues on next page)

(continued from previous page)

```
    tmpcptr = transfer(u_descptr%data(1), tmpcptr)
    call c_f_pointer(tmpcptr, u_dptr, local_permuted_cshape(1:3))

    call scalingData(local_permuted_cshape(1), local_permuted_cshape(2), &
        local_permuted_cshape(3), u_dptr, real(nx*ny*nz))
#else
    call c_f_pointer(u_descptr%data(1), u_dptr, local_permuted_cshape(1:3))
    !$cuf kernel do (3)
    do k =1, local_permuted_cshape(3)
        do j = 1, local_permuted_cshape(2)
            do i = 1, local_permuted_cshape(1)
                u_dptr(i,j,k) = u_dptr(i,j,k) / real(nx*ny*nz)
            end do
        end do
    end do
    call checkCuda(cudaDeviceSynchronize())
#endif

    ! in case we want to check again after scaling
    call checkCufft(cufftXtMemcpy(planr2c, u_permuted, u_desc, &
        CUFFT_COPY_DEVICE_TO_HOST), 'permuted D2H error')
    call checkNormComplex(local_permuted_cshape(1), &
        local_permuted_cshape(2), local_permuted_cshape(3), &
        u_permuted, max_norm)
    write(*,'(A18, I1, A14, F25.8)') "after scaling ", rank, &
        " max_norm is ", max_norm

    !xxxxxxxxxxxxxxxxxxxxxxxxxxxx inverse
    call checkCufft(cufftXtExecDescriptor(planc2r, u_desc, u_desc, &
        CUFFT_INVERSE), 'inverse fft failed')

    ! These are equivalent, and work as well
    ! istat = cufftXtMemcpy(planc2r, u, u_desc, CUFFT_COPY_DEVICE_TO_HOST)
    ! call cufft_memcpyD2H(u, u_desc, CUFFT_XT_FORMAT_INPLACE, .false.)
    call cufft_memcpyD2H(u, u_desc, CUFFT_XT_FORMAT_INPLACE, .true.)

    call checkCufft(cufftXtFree(u_desc))
    call checkCufft(cufftDestroy(planr2c))
    call checkCufft(cufftDestroy(planc2r))

    call checkNormDiff(nz, local_rshape(1), local_rshape(2), &
        local_rshape(3), u, ref, max_norm, max_diff)
    write(*,'(A18,I1,A14,F25.8,A14,F15.8)') "after C2R ", rank, &
        " max_norm is ", max_norm, " max_diff is ", max_diff
    write(*,'(A25,I1,A14,F25.8)') "Relative Linf on rank ", rank, &
        " is ", max_diff/max_norm
    deallocate(u)
    deallocate(ref)
    deallocate(u_permuted)

    call mpi_finalize(ierr)

    if(max_diff / max_norm > 1e-5) then
        print*, ">>>> FAILED on rank ", rank
        stop 1
    else
```

(continues on next page)

(continued from previous page)

```
        print*, ">>>> PASSED on rank ", rank
    end if

contains
#ifdef _CUDA
    subroutine checkCuda(istat, message)
        implicit none
        integer, intent(in)                    :: istat
        character(len=*),intent(in), optional :: message
        if (istat /= cudaSuccess) then
            write(*,"('Error code: ',I0, ': ')") istat
            write(*,*) cudaGetErrorString(istat)
            if(present(message)) write(*,*) message
            call mpi_finalize(ierr)
        endif
    end subroutine checkCuda
#endif

    subroutine checkCufft(istat, message)
        implicit none
        integer, intent(in)                    :: istat
        character(len=*),intent(in), optional :: message
        if (istat /= CUFFT_SUCCESS) then
            write(*,"('Error code: ',I0, ': ')") istat
!@cuf       write(*,*) cudaGetErrorString(istat)
            if(present(message)) write(*,*) message
            call mpi_finalize(ierr)
        endif
    end subroutine checkCufft

    subroutine generate_random(nz1, nz, ny, nx, data)
        implicit none
        integer, intent(in) :: nx, ny, nz, nz1
        real, dimension(nz, ny, nx), intent(out) :: data
        real :: rand(1)
        integer :: i,j,k
        !call random_seed(put=(/seed, seed+1/))
        do k =1, nx
            do j = 1, ny
                do i = 1, nz1
                    call random_number(rand)
                    data(i,j,k) = rand(1)
                end do
            end do
        end do

    end subroutine generate_random

    subroutine checkNorm(nz1, nz, ny, nx, data, max_norm)
        implicit none
        integer, intent(in)  :: nx, ny, nz, nz1
        real, dimension(nz, ny, nx), intent(in) :: data
        real :: max_norm
        integer :: i, j, k
        max_norm = 0
        do k =1, nx
```

(continues on next page)

(continued from previous page)

```
            do j = 1, ny
                do i = 1, nz1
                    max_norm = max(max_norm, abs(data(i,j,k)))
                end do
            end do
        end do
    end subroutine checkNorm

    subroutine checkNormComplex(nz, ny, nx, data, max_norm)
        implicit none
        integer, intent(in)  :: nx, ny, nz
        complex, dimension(nz, ny, nx), intent(in) :: data
        real :: max_norm, max_diff
        integer :: i,j,k
        max_norm = 0
        do k =1, nx
            do j = 1, ny
                do i = 1, nz
                    max_norm = max(max_norm, abs(data(i,j,k)%re))
                    max_norm = max(max_norm, abs(data(i,j,k)%im))
                end do
            end do
        end do
    end subroutine checkNormComplex

    subroutine checkNormDiff(nz1, nz, ny, nx, data, ref, max_norm, max_diff)
        implicit none
        integer, intent(in)  :: nx, ny, nz, nz1
        real, dimension(nz, ny, nx), intent(in) :: data, ref
        real :: max_norm, max_diff
        max_norm = 0
        max_diff = 0
        do k =1, nx
          do j = 1, ny
            do i = 1, nz1
              max_norm = max(max_norm, abs(data(i,j,k)))
              max_diff = max(max_diff, abs(ref(i,j,k)-data(i,j,k)))
              if (abs(ref(i,j,k)-data(i,j,k)) > 0.0001) then
                write(*,'(A9,I3,I3,I3,A2,F18.8,A7,F18.8,A9,I2)') "diff ref[", &
                  i,j,k,"]",ref(i,j,k),"data ",data(i,j,k)," at rank",rank
              end if
            end do
          end do
        end do
    end subroutine checkNormDiff

#ifdef _OPENACC
    subroutine scalingData(nz, ny, nx, data, factor)
        implicit none
        integer, intent(in)  :: nx, ny, nz
        complex, dimension(nz, ny, nz) :: data
        !$acc declare deviceptr(data)
        real, intent(in)     :: factor

    !$acc parallel loop collapse(3)
        do k =1, nx
```

(continues on next page)

(continued from previous page)

```
            do j = 1, ny
                do i = 1, nz
                    data(i, j, k) = data(i, j, k) / factor
                end do
            end do
        end do

    end subroutine scalingData
#endif

subroutine cufft_memcpyH2D(ulibxt, u_h, data_format, ismemcpy)
    implicit none
    type(cudaLibXtDesc), pointer, intent(out) :: ulibxt
    real, dimension(*), intent(in)          :: u_h
    integer, intent(in)                         :: data_format
    logical, intent(in)                         :: ismemcpy
    type(cudaXtDesc), pointer  :: uxt
!@cuf real, dimension(:,:,:), device, pointer :: u_d

    if (ismemcpy == .false.) then
        call checkCufft(cufftXtMemcpy(planc2r, ulibxt, u_h, &
                 CUFFT_COPY_HOST_TO_DEVICE), "cufft_memcpyHToD Error")
    else
        call c_f_pointer(ulibxt%descriptor, uxt)
        if(data_format == CUFFT_XT_FORMAT_INPLACE_SHUFFLED) then
#ifdef _OPENACC
            call acc_memcpy_to_device(uxt%data(1), u_h, &
                     product(int(local_rshape_permuted,kind=8))*4_8) ! bytes
#else
            call c_f_pointer(uxt%data(1), u_d, local_rshape_permuted)
            call checkCuda(cudaMemcpy(u_d, u_h, &
                     product(int(local_rshape_permuted,kind=8))), &
                     "cudamemcpy H2D Error")
#endif
        else if (data_format == CUFFT_XT_FORMAT_INPLACE) then
#ifdef _OPENACC
            call acc_memcpy_to_device(uxt%data(1), u_h, &
                     product(int(local_rshape,kind=8))*4_8) ! bytes
#else
            call c_f_pointer(uxt%data(1), u_d, local_rshape)
            call checkCuda(cudaMemcpy(u_d, u_h, &
                     product(int(local_rshape,kind=8))), &
                     "cudamemcpy H2D Error")
#endif
        endif
    endif
end subroutine cufft_memcpyH2D


subroutine cufft_memcpyD2H(u_h, ulibxt, data_format,ismemcpy)
    implicit none
    type(cudaLibXtDesc), pointer, intent(in) :: ulibxt
    real, dimension(*), intent(out)      :: u_h
    integer, intent(in)                      :: data_format
    logical, intent(in)                      :: ismemcpy
    type(cudaXtDesc), pointer  :: uxt
```

(continues on next page)

(continued from previous page)

```
!@cuf real, dimension(:,:,:), device, pointer :: u_d

    if (ismemcpy == .false.) then
        call checkCufft(cufftXtMemcpy(planr2c, u_h, ulibxt, &
                 CUFFT_COPY_DEVICE_TO_HOST), "cufft_memcpyDToH Error")
    else
        call c_f_pointer(ulibxt%descriptor, uxt)
        if(data_format == CUFFT_XT_FORMAT_INPLACE_SHUFFLED) then
#ifdef _OPENACC
            call acc_memcpy_from_device(u_h, uxt%data(1), &
                     product(int(local_rshape_permuted,kind=8))*4_8) ! bytes
#else
            call c_f_pointer(uxt%data(1), u_d, local_rshape_permuted)
            call checkCuda(cudaMemcpy(u_h, u_d, &
                     product(int(local_rshape_permuted,kind=8))), &
                     "cudamemcpy D2H Error")
#endif
        else if (data_format == CUFFT_XT_FORMAT_INPLACE) then
#ifdef _OPENACC
            call acc_memcpy_from_device(u_h, uxt%data(1), &
                     product(int(local_rshape,kind=8))*4_8) ! bytes
#else
            call c_f_pointer(uxt%data(1), u_d, local_rshape)
            call checkCufft(cudamemcpy(u_h, u_d, &
                     product(int(local_rshape,kind=8))), &
                     "cufft_memcpyD2H error")
#endif
        endif
    endif
end subroutine cufft_memcpyD2H


end program cufftmp_r2c
```

19.7. Using cuRAND from OpenACC Host Code

This example demonstrates the use of the curand module, the curandHandle type, and several forms of rand calls.

Simple cuRAND Tests

```
program testcurand
! compile with the flags -ta=tesla -cuda -cudalib=curand
call cur1(1000, .true.); call cur1(1000, .false.)
call cur2(1000, .true.); call cur2(1000, .false.)
call cur3(1000, .true.); call cur3(1000, .false.)
end
!
subroutine cur1(n, onhost)
use curand
integer :: a(n)
type(curandGenerator) :: g
integer(8) nbits
```

(continues on next page)

(continued from previous page)

```
logical onhost, passing
a = 0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
nbits = 0
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  nbits = nbits + popcnt(a(i))
end do
print *,"Should be roughly half the bits set"
nbits = nbits / n
if ((nbits .lt. 12) .or. (nbits .gt. 20)) then
  passing = .false.
else
  print *,"nbits is ",nbits," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine cur2(n, onhost)
use curand
real :: a(n)
type(curandGenerator) :: g
logical onhost, passing
a = 0.0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
print *,"Should be uniform around 0.5"
```

(continues on next page)

(continued from previous page)

```
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0) .or. (a(i).gt.1.0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine cur3(n, onhost)
use curand
real(8) :: a(n)
type(curandGenerator) :: g
logical onhost, passing
a = 0.0d0
passing = .true.
if (onhost) then
  istat = curandCreateGeneratorHost(g,CURAND_RNG_PSEUDO_XORWOW)
  istat = curandGenerate(g, a, n)
  istat = curandDestroyGenerator(g)
else
  !$acc data copy(a)
  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, n)
  !$acc end host_data
  istat = curandDestroyGenerator(g)
  !$acc end data
endif
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0d0) .or. (a(i).gt.1.0d0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4d0) .or. (rmean .gt. 0.6d0)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
```

19.8. Using cuRAND from OpenACC Device Code

This example demonstrates the use of the curand_device module from a CUDA Fortran global subroutines.

Simple cuRAND Test from OpenACC Device Code

```
module mtests
  integer, parameter :: n = 1000
  contains
    subroutine testrand( a, b )
    use openacc_curand
    real :: a(n), b(n)
    type(curandStateXORWOW) :: h
    integer(8) :: seed, seq, offset

    !$acc parallel num_gangs(1) vector_length(1) copy(a,b) private(h)
    seed = 12345
    seq = 0
    offset = 0
    call curand_init(seed, seq, offset, h)
    !$acc loop seq
    do i = 1, n
      a(i) = curand_uniform(h)
      b(i) = curand_normal(h)
    end do
    !$acc end parallel
    return
    end subroutine
end module mtests

program t
use mtests
real :: a(n), b(n), c(n)
logical passing
a = 1.0
b = 2.0
passing = .true.
call testrand(a,b)
c = a
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i).lt.0.0) .or. (c(i).gt.1.0)) passing = .false.
end do
rmean = sum(c)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
c = b
print *,"Should be normal around 0.0"
nc1 = 0;
```

(continues on next page)

(continued from previous page)

```
nc2 = 0;
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i) .gt. -4.0) .and. (c(i) .lt. 0.0)) nc1 = nc1 + 1
  if ((c(i) .gt.  0.0) .and. (c(i) .lt. 4.0)) nc2 = nc2 + 1
end do
print *,"Found on each side of zero ",nc1,nc2
if (abs(nc1-nc2) .gt. (n/10)) npassing = .false.
rmean = sum(c,mask=abs(c).lt.4.0)/n
if ((rmean .lt. -0.1) .or. (rmean .gt. 0.1)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end program
```

19.9. Using cuRAND from CUDA Fortran Host Code

This example demonstrates the use of the curand module, the curandHandle type, and several forms of rand calls.

Simple cuRAND Test

```
program testcurand1
call testr1(1000)
call testr2(1000)
call testr3(1000)
end
!
subroutine testr1(n)
use cudafor
use curand
integer, managed :: a(n)
type(curandGenerator) :: g
integer(8) nbits
logical passing
a = 0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
nbits = 0
do i = 1, n
  if (i.lt.10) print *,i,a(i)
```

(continues on next page)

(continued from previous page)

```
  nbits = nbits + popcnt(a(i))
end do
print *,"Should be roughly half the bits set"
nbits = nbits / n
if ((nbits .lt. 12) .or. (nbits .gt. 20)) then
  passing = .false.
else
  print *,"nbits is ",nbits," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testr2(n)
use cudafor
use curand
real, managed :: a(n)
type(curandGenerator) :: g
logical passing
a = 0.0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0) .or. (a(i).gt.1.0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
!
subroutine testr3(n)
use cudafor
use curand
real(8), managed :: a(n)
type(curandGenerator) :: g
logical passing
a = 0.0d0
passing = .true.
istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)
istat = curandGenerate(g, a, n)
```

(continues on next page)

(continued from previous page)

```
istat = cudaDeviceSynchronize()
istat = curandDestroyGenerator(g)
do i = 1, n
  if (i.lt.10) print *,i,a(i)
  if ((a(i).lt.0.0d0) .or. (a(i).gt.1.0d0)) passing = .false.
end do
rmean = sum(a)/n
if ((rmean .lt. 0.4d0) .or. (rmean .gt. 0.6d0)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif
if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
end program
```

19.10. Using cuRAND from CUDA Fortran Device Code

This example demonstrates the use of the curand_device module from a CUDA Fortran global subroutines.

Simple cuRAND Test from Device Code

```
module mtests
  use curand_device
  integer, parameter :: n = 10000
  contains
    attributes(global) subroutine testr( a, b )
    real, device :: a(n), b(n)
    type(curandStateXORWOW) :: h
    integer(8) :: seed, seq, offset
    integer :: iam
    iam = threadIdx%x
    seed = iam*64 + 12345
    seq = 0
    offset = 0
    call curand_init(seed, seq, offset, h)
    do i = iam, n, blockdim%x
      a(i) = curand_uniform(h)
      b(i) = curand_normal(h)
    end do
    return
    end subroutine
end module mtests

program t
use mtests
```

(continues on next page)

(continued from previous page)

```
real, allocatable, device :: a(:), b(:)
real c(n), rmean
logical passing
allocate(a(n))
allocate(b(n))
a = 0.0
b = 0.0
passing = .true.
call testr<<<1,32>>> (a,b)
c = a
print *,"Should be uniform around 0.5"
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i).lt.0.0) .or. (c(i).gt.1.0)) passing = .false.
end do
rmean = sum(c)/n
if ((rmean .lt. 0.4) .or. (rmean .gt. 0.6)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

c = b
print *,"Should be normal around 0.0"
nc1 = 0;
nc2 = 0;
do i = 1, n
  if (i.lt.10) print *,i,c(i)
  if ((c(i) .gt. -4.0) .and. (c(i) .lt. 0.0)) nc1 = nc1 + 1
  if ((c(i) .gt.  0.0) .and. (c(i) .lt. 4.0)) nc2 = nc2 + 1
end do
print *,"Found on each side of zero ",nc1,nc2
if (abs(nc1-nc2) .gt. (n/10)) npassing = .false.
rmean = sum(c,mask=abs(c).lt.4.0)/n
if ((rmean .lt. -0.1) .or. (rmean .gt. 0.1)) then
  passing = .false.
else
  print *,"Mean is ",rmean," which passes"
endif

if (passing) then
  print *,"Test PASSED"
else
  print *,"Test FAILED"
endif
end
end program
```

19.11. Using cuSPARSE from OpenACC Host Code

This example demonstrates the use of the cuSPARSE module, the cusparseHandle type, and several calls to the cuSPARSE library.

Simple BLAS Test

```
program sparseMatVec
    integer n
    n = 25 ! # rows/cols in dense matrix
    call sparseMatVecSub1(n)
    n = 45 ! # rows/cols in dense matrix
    call sparseMatVecSub1(n)
end program

subroutine sparseMatVecSub1(n)
  use openacc
  use cusparse

  implicit none

  integer n

  ! dense data
  real(4), allocatable :: Ade(:,:), x(:), y(:)

  ! sparse CSR arrays
  real(4), allocatable :: csrValA(:)
  integer, allocatable :: nnzPerRowA(:), csrRowPtrA(:), csrColIndA(:)

  allocate(Ade(n,n), x(n), y(n))
  allocate(csrValA(n))
  allocate(nnzPerRowA(n), csrRowPtrA(n+1), csrColIndA(n))

  call sparseMatVecSub2(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, &
                                                        csrColIndA, n)
  deallocate(Ade)
  deallocate(x)
  deallocate(y)
  deallocate(csrValA)
  deallocate(nnzPerRowA)
  deallocate(csrRowPtrA)
  deallocate(csrColIndA)
end subroutine

subroutine sparseMatVecSub2(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, &
                                                          csrColIndA, n)
  use openacc
  use cusparse

  implicit none

  ! dense data
  real(4) :: Ade(n,n), x(n), y(n)
```

(continues on next page)

(continued from previous page)

```
! sparse CSR arrays
real(4) :: csrValA(n)
integer :: nnzPerRowA(n), csrRowPtrA(n+1), csrColIndA(n)

integer :: n, nnz, status, i
type(cusparseHandle) :: h
type(cusparseMatDescr) :: descrA

! parameters
real(4) :: alpha, beta

! result
real(4) :: xerr

! initalize CUSPARSE and matrix descriptor
status = cusparseCreate(h)
if (status /= CUSPARSE_STATUS_SUCCESS) &
     write(*,*) 'cusparseCreate error: ', status
status = cusparseCreateMatDescr(descrA)
status = cusparseSetMatType(descrA, &
     CUSPARSE_MATRIX_TYPE_GENERAL)
status = cusparseSetMatIndexBase(descrA, &
     CUSPARSE_INDEX_BASE_ONE)
status = cusparseSetStream(h, acc_get_cuda_stream(acc_async_sync))

!$acc data create(Ade, x, y, csrValA, nnzPerRowA, csrRowPtrA, csrColIndA)

! Initialize matrix (upper circular shift matrix)
!$acc kernels
Ade = 0.0
do i = 1, n-1
   Ade(i,i+1) = 1.0
end do
Ade(n,1) = 1.0

! Initialize vectors and constants
do i = 1, n
   x(i) = i
enddo
y = 0.0
!$acc end kernels

!$acc update host(x)
write(*,*) 'Original vector:'
write(*,'(5(1x,f7.2))') x

! convert matrix from dense to csr format
!$acc host_data use_device(Ade, nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)
status = cusparseSnnz_v2(h, CUSPARSE_DIRECTION_ROW, &
     n, n, descrA, Ade, n, nnzPerRowA, nnz)
status = cusparseSdense2csr(h, n, n, descrA, Ade, n, &
     nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)
!$acc end host_data

! A is upper circular shift matrix
```

(continues on next page)

(continued from previous page)

```
  ! y = alpha*A*x + beta*y
  alpha = 1.0
  beta = 0.0
  !$acc host_data use_device(csrValA, csrRowPtrA, csrColIndA, x, y)
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
       n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
       csrColIndA, x, beta, y)
  !$acc end host_data

  !$acc wait
  write(*,*) 'Shifted vector:'
  write(*,'(5(1x,f7.2))') y

  ! shift-down y and add original x
  ! A' is lower circular shift matrix
  ! x = alpha*A'*y + beta*x
  beta = -1.0
  !$acc host_data use_device(csrValA, csrRowPtrA, csrColIndA, x, y)
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_TRANSPOSE, &
       n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
       csrColIndA, y, beta, x)
  !$acc end host_data

  !$acc kernels
  xerr = maxval(abs(x))
  !$acc end kernels
  !$acc end data

  write(*,*) 'Max error = ', xerr
  if (xerr.le.1.e-5) then
    write(*,*) 'Test PASSED'
  else
    write(*,*) 'Test FAILED'
  endif

end subroutine
```

19.12. Using cuSPARSE from CUDA Fortran Host Code

This example demonstrates the use of the cuSPARSE module, the cusparseHandle type, and several forms of cusparse calls.

Simple BLAS Test

```
program sparseMatVec
  use cudafor
  use cusparse

  implicit none

  integer, parameter :: n = 25 ! # rows/cols in dense matrix
```

(continues on next page)

(continued from previous page)

```
type(cusparseHandle) :: h
type(cusparseMatDescr) :: descrA

! dense data
real(4), managed :: Ade(n,n), x(n), y(n)

! sparse CSR arrays
real(4), managed :: csrValA(n)
integer, managed :: nnzPerRowA(n), &
     csrRowPtrA(n+1), csrColIndA(n)
integer :: nnz, status, i

! parameters
real(4) :: alpha, beta

! initalize CUSPARSE and matrix descriptor
status = cusparseCreate(h)
if (status /= CUSPARSE_STATUS_SUCCESS) &
     write(*,*) 'cusparseCreate error: ', status
status = cusparseCreateMatDescr(descrA)
status = cusparseSetMatType(descrA, &
     CUSPARSE_MATRIX_TYPE_GENERAL)
status = cusparseSetMatIndexBase(descrA, &
     CUSPARSE_INDEX_BASE_ONE)

! Initialize matrix (upper circular shift matrix)
Ade = 0.0
do i = 1, n-1
   Ade(i,i+1) = 1.0
end do
Ade(n,1) = 1.0

! Initialize vectors and constants
x = [(i,i=1,n)]
y = 0.0

write(*,*) 'Original vector:'
write(*,'(5(1x,f7.2))') x

! convert matrix from dense to csr format
status = cusparseSnnz_v2(h, CUSPARSE_DIRECTION_ROW, &
     n, n, descrA, Ade, n, nnzPerRowA, nnz)
status = cusparseSdense2csr(h, n, n, descrA, Ade, n, &
     nnzPerRowA, csrValA, csrRowPtrA, csrColIndA)

! A is upper circular shift matrix
! y = alpha*A*x + beta*y
alpha = 1.0
beta = 0.0
status = cusparseScsrmv(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
     n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
     csrColIndA, x, beta, y)

! shift-down y and add original x
! A' is lower circular shift matrix
```

(continues on next page)

(continued from previous page)

```
  ! x = alpha*A'*y + beta*x
  beta = -1.0
  status = cusparseScsrmv(h, CUSPARSE_OPERATION_TRANSPOSE, &
       n, n, n, alpha, descrA, csrValA, csrRowPtrA, &
       csrColIndA, y, beta, x)

  status = cudaDeviceSynchronize()

  write(*,*) 'Shifted vector:'
  write(*,'(5(1x,f7.2))') y

  write(*,*) 'Max error = ', maxval(abs(x))

  if (maxval(abs(x)).le.1.e-5) then
    write(*,*) 'Test PASSED'
  else
    write(*,*) 'Test FAILED'
  endif

end program sparseMatVec
```

19.13. Using cuTENSOR from CUDA Fortran Host Code

This example demonstrates the use of the low-level cuTENSOR module, version 2, from CUDA Fortran to perform a reshape permutation and a sum reduction across one dimension.

This example can be compiled and linked as a normal CUDA Fortran subprogram by adding the "-cudalib=cutensor" option to the link line.

Simple cuTENSOR Test from CUDA Fortran

```
program testcutensorcuf1
use cudafor
use cutensor_v2

integer, parameter :: ndim = 3
real, managed, allocatable :: dA(:,:,:), dC(:,:)
real, allocatable :: hA(:,:,:), hC(:,:)
real, device, allocatable :: workbuf(:)
real :: alpha, beta
integer(4) :: numModesA, numModesC
integer(8), dimension(ndim) :: extA, strA, extC, strC
integer(4), dimension(ndim) :: modeA, modeC
integer(8) :: workbufsize

type(cutensorStatus) :: cstat
type(cutensorHandle) :: handle
type(cutensorTensorDescriptor) :: descA, descC
type(cutensorOperationDescriptor) :: rdesc
type(cutensorComputeDescriptor) :: descCompute
type(cutensorPlan) :: plan
```

(continues on next page)

(continued from previous page)

```
type(cutensorAlgo) :: algo
type(cutensorPlanPreference) :: pref

! Init
cstat = cutensorCreate(handle)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

descCompute = CUTENSOR_COMPUTE_DESC_32F

allocate(dA(100,60,80))
! This is the operation we are going to perform
! dC = sum(reshape(dA,shape=[80,60,100],order=[3,2,1]),dim=2)
!
call random_number(dA); dA = real(int(dA * 10.0))
extA = shape(dA)
strA(1) =  1; strA(2) = 100; strA(3) = 6000
modeA(1) = 3; modeA(2) = 2; modeA(3) = 1
numModesA = ndim
ialign = 256

print *,"Desc A"
cstat = cutensorCreateTensorDescriptor(handle, descA, numModesA, extA, strA, &
              CUTENSOR_R_32F, ialign)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(dC(80,100))
extC(1:ndim-1) = shape(dC)
strC(1) = 1; strC(2) = 80
numModesC = ndim-1
dC = 0.0
modeC(1) = 1; modeC(2) = 3  ! Skip mode 2 for reduction across that dim

print *,"Desc C"
cstat = cutensorCreateTensorDescriptor(handle, descC, numModesC, extC, strC, &
              CUTENSOR_R_32F, ialign)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

print *,"CreateRed "
cstat = cutensorCreateReduction(handle, rdesc, &
              descA, modeA, CUTENSOR_OP_IDENTITY, &
              descC, modeC, CUTENSOR_OP_IDENTITY, &
              descC, modeC, CUTENSOR_OP_ADD, descCompute)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

print *,"CreatePlanPref "
cstat = cutensorCreatePlanPreference(handle, pref, CUTENSOR_ALGO_DEFAULT, &
              CUTENSOR_JIT_MODE_NONE)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

print *,"EstimateWork "
cstat = cutensorEstimateWorkspaceSize(handle, rdesc, pref, &
              CUTENSOR_WORKSPACE_DEFAULT, workbufsize)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

print *,"Estimated workspace size: ",workbufsize
```

(continues on next page)

(continued from previous page)

```
print *,"CreatePlan "
cstat = cutensorCreatePlan(handle, plan, rdesc, pref, workbufsize)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(workbuf((workbufsize+3)/4))

alpha = 1.0; beta = 0.0
print *,"Reduce "
cstat = cutensorReduce(handle, plan, alpha, dA, beta, dC, &
             dC, workbuf, workbufsize, 0)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

hA = dA
hC = sum(reshape(hA,[80,60,100],order=[3,2,1]), dim=2)
istat = cudaDeviceSynchronize()  ! Managed memory, to be sure

if (all(hC.eq.dC)) then
  print *,"test PASSED"
else
  print *,"test FAILED"
end if

cstat = cutensorDestroy(handle)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)
cstat = cutensorDestroyPlan(plan)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)
cstat = cutensorDestroyPlanPreference(pref)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)
cstat = cutensorDestroyOperationDescriptor(rdesc)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)
cstat = cutensorDestroyTensorDescriptor(descA)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)
cstat = cutensorDestroyTensorDescriptor(descC)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

deallocate(workbuf)
deallocate(dA, dC)
deallocate(hA, hC)

end program
```

19.14. Using cuTENSOREX from CUDA Fortran Host Code

This example demonstrates the use of the higher-level cuTENSOREX module from CUDA Fortran to perform a large matrix multiplication using multiple OpenMP threads.

This example can be compiled and linked as a normal CUDA Fortran subprogram by adding the “-mp -cudalib=cutensor” options to the compile and link line. Increase the default OpenMP stacksize when executing this test (ex. OMP_STACKSIZE=16M).

Multi-threaded cuTENSOREX Example from CUDA Fortran

```
! Test cuTensor + cuda Fortran + OMP multi-stream matmul
program testCuCufMsMatmul
use cudafor
use cutensorex

integer, parameter :: m=8192, k=1280, n=1024
integer, parameter :: mblksize = 128
integer, parameter :: mslices = m / mblksize
integer, parameter :: nstreams = 4
integer, parameter :: numtimes = mslices / nstreams

real(8), allocatable, dimension(:,:), device :: a_d, d_d
real(8), allocatable, dimension(:,:,:), pinned :: ha
real(8), dimension(k,n), device :: b_d
real(8), allocatable, dimension(:,:), pinned :: d
real(8) :: alpha
integer(kind=cuda_stream_kind) :: mystream
!$OMP THREADPRIVATE(a_d, d_d, mystream)

allocate( ha(k,mblksize,nstreams))
allocate(  d(1:m,1:n))

b_d   = 1.0d0
alpha = 1.0d0

!$OMP PARALLEL NUM_THREADS(nstreams) PRIVATE(istat)
istat = cudaStreamCreate(mystream)
istat = cudaforSetDefaultStream(mystream)
istat = cutensorExSetStream(mystream)
! At this point, all new allocations will pick up default stream
allocate(a_d(k,mblksize))
allocate(d_d(mblksize,n))
!$OMP END PARALLEL

! Test matmul
!$OMP PARALLEL DO NUM_THREADS(nstreams) PRIVATE(jlcl,jgbl,jend)
do ns = 1, nstreams
    do nt = 1, numtimes
        jgbl = 1 + ((ns-1) + (nt-1)*nstreams)*mblksize
        jend = jgbl + mblksize - 1
        ! Do some host work
        do jlcl = 1, mblksize
            ha(:,jlcl,ns) = dble(jlcl+jgbl-1)
        end do
        ! Move data to the device on default stream
        a_d = ha(:,:,ns)
        ! Matrix multiply on my thread cutensorEx stream
        d_d = alpha * matmul(transpose(a_d),b_d)
        ! Move result back to host on default stream
        d(jgbl:jend,:) = d_d
    end do
end do
! Wait for all threads to finish GPU work
istat = cudaDeviceSynchronize()
nfailed = 0
do j = 1, n
    do i = 1, m
```

(continues on next page)

(continued from previous page)

```
        if (d(i,j) .ne. i*k) then
            if (nfailed .lt. 100) print *,i,j,d(i,j)
            nfailed = nfailed + 1
        end if
    end do
end do
if (nfailed .eq. 0) then
    print *,"test PASSED"
else
    print *,"test FAILED"
endif
end program
```

19.15. Using cuTENSOR from OpenACC Host Code

This example demonstrates the use of the cuTENSOREX module, calling Matmul() using OpenACC device data, and setting the cuTENSOR library stream to be consistent with the OpenACC default stream.

This example can be compiled and run with or without OpenACC. To compile with OpenACC, the options are "-ta=tesla -cuda -cudalib=cutensor". To run on the CPU, leave off those options.

Simple cuTENSOREX Test from OpenACC

```
program testcutensorOpenACC
!@acc use openacc
!@acc use cutensorex
integer, parameter :: ni=1280, nj=1024, nk=960, ntimes=1
real(8) :: a(ni,nk), b(nk,nj), c(ni,nj), d(ni,nj)

call random_number(a)
call random_number(b)
a = dble(int(4.0d0*a - 2.0d0))
b = dble(int(8.0d0*b - 4.0d0))
c = 2.0; d = 0.0

!$acc enter data copyin(a,b,c) create(d)
!@acc istat = cutensorExSetStream(acc_get_cuda_stream(acc_async_sync))
!$acc host_data use_device(a,b,c,d)
do nt = 1, ntimes
  d = c + matmul(a,b)
end do
!$acc end host_data

!$acc update host(d)
print *,sum(d)

do nt = 1, ntimes
!$acc kernels
  do j = 1, nj
    do i = 1, ni
```

(continues on next page)

(continued from previous page)

```
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
!$acc end kernels
end do
!$acc exit data copyout(d)

print *,sum(d)
end program
```

Notices

Notice and Disclaimers

All information provided in this document is provided as-is, for your informational purposes only and is subject to change at any time without notice. Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing. To obtain the latest information, please contact your NVIDIA representative. Product or service performance varies by use, configuration and other factors. Your costs and results may vary. No product or component is absolutely secure. TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, NVIDIA DISCLAIMS ALL WARRANTIES AND REPRESENTATIONS OF ANY KIND, WHETHER EXPRESS, IMPLIED OR STATUTORY, RELATING TO OR ARISING UNDER THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, THE WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, USAGE OF TRADE AND COURSE OF DEALING. NVIDIA products are not intended or authorized for use as critical components in a system or application where the use of or failure of such system or application developed with products, technology, software or services provided by NVIDIA could result in injury, death or catastrophic damage.

Except for your permitted use of the information contained in this document, no license or right is granted by implication, estoppel or otherwise. If this document directly includes or links to third-party websites, products, services or information, please consult other sources to evaluate if and how to use that information since NVIDIA does not support, endorse or assume any responsibility for any third party offerings or its accuracy or usefulness.

TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY (I) INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, OR (II) DAMAGES FOR THE (A) COST OF PROCURING SUBSTITUTE GOODS OR (B) LOSS OF PROFITS, REVENUES, USE, DATA OR GOODWILL ARISING OUT OF OR RELATED TO THIS DOCUMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, OR OTHERWISE, AND EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES AND EVEN IF A PARTY'S REMEDIES FAIL THEIR ESSENTIAL PURPOSE. ADDITIONALLY, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, NVIDIA'S TOTAL CUMULATIVE AGGREGATE LIABILITY FOR ANY AND ALL LIABILITIES, OBLIGATIONS OR CLAIMS ARISING OUT OF OR RELATED TO THIS DOCUMENT WILL NOT EXCEED FIVE U.S. DOLLARS (US$5).

Statements in this document that refer to future plans or expectations are forward-looking statements. These statements are based on currently available information, beliefs, assumptions and involve many risks and uncertainties that could cause actual results to differ materially from those expressed or implied in these statements. For more information on the factors that could cause actual results to differ materially, see our most recent earnings release and SEC filings at NVIDIA Corporation SEC Filings.

© NVIDIA Corporation. All rights reserved. NVIDIA, the NVIDIA logo, and other NVIDIA marks are trademarks of NVIDIA Corporation or its affiliates. Other names and brands may be claimed as the property of others.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, CUDA-X, GPUDirect, HPC SDK, NGC, NVIDIA Volta, NVIDIA DGX, NVIDIA Nsight, NVLink, NVSwitch, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

©2022-2026, NVIDIA Corporation & affiliates. All rights reserved