



# NVIDIA<sup>®</sup>

## HPC Compilers

FORTRAN CUDA INTERFACES

# TABLE OF CONTENTS

Preface.....	xxxix
Intended Audience.....	xxxix
Organization.....	xxxix
Conventions.....	xl
Terminology.....	xl
Related Publications.....	xl
Chapter 1. Introduction.....	1
1.1. Fortran Interfaces and Wrappers.....	2
1.2. Using CUDA Libraries from OpenACC Host Code.....	2
1.3. Using CUDA Libraries from OpenACC Device Code.....	3
1.4. Using CUDA Libraries from CUDA Fortran Host Code.....	4
1.5. Using CUDA Libraries from CUDA Fortran Device Code.....	5
1.6. Pointer Modes in cuBLAS and cuSPARSE.....	5
1.7. Writing Your Own CUDA Interfaces.....	6
1.8. NVIDIA Fortran Compiler Options.....	8
Chapter 2. BLAS Runtime APIs.....	10
2.1. CUBLAS Definitions and Helper Functions.....	11
2.1.1. cublasCreate.....	12
2.1.2. cublasDestroy.....	12
2.1.3. cublasGetVersion.....	12
2.1.4. cublasSetStream.....	12
2.1.5. cublasGetStream.....	13
2.1.6. cublasGetStatusName.....	13
2.1.7. cublasGetStatusString.....	13
2.1.8. cublasGetPointerMode.....	13
2.1.9. cublasSetPointerMode.....	13
2.1.10. cublasGetAtomicsMode.....	14
2.1.11. cublasSetAtomicsMode.....	14
2.1.12. cublasGetMathMode.....	14
2.1.13. cublasSetMathMode.....	14
2.1.14. cublasGetHandle.....	14
2.1.15. cublasSetVector.....	14
2.1.16. cublasGetVector.....	15
2.1.17. cublasSetMatrix.....	15
2.1.18. cublasGetMatrix.....	15
2.1.19. cublasSetVectorAsync.....	15
2.1.20. cublasGetVectorAsync.....	16
2.1.21. cublasSetMatrixAsync.....	16
2.1.22. cublasGetMatrixAsync.....	16
2.2. Single Precision Functions and Subroutines.....	17

2.2.1. isamax.....	17
2.2.2. isamin.....	17
2.2.3. sasum.....	17
2.2.4. saxpy.....	18
2.2.5. scopy.....	18
2.2.6. sdot.....	18
2.2.7. snrm2.....	19
2.2.8. srot.....	19
2.2.9. srotg.....	19
2.2.10. srotm.....	20
2.2.11. srotmg.....	20
2.2.12. sscal.....	20
2.2.13. sswap.....	21
2.2.14. sgbmv.....	21
2.2.15. sgemv.....	22
2.2.16. sger.....	22
2.2.17. ssbmv.....	22
2.2.18. sspmv.....	23
2.2.19. sspr.....	23
2.2.20. sspr2.....	24
2.2.21. ssymv.....	24
2.2.22. ssyr.....	24
2.2.23. ssyr2.....	25
2.2.24. stbmv.....	25
2.2.25. stbsv.....	26
2.2.26. stpmv.....	26
2.2.27. stpsv.....	26
2.2.28. strmv.....	27
2.2.29. strsv.....	27
2.2.30. sgemm.....	28
2.2.31. ssymm.....	28
2.2.32. ssyrk.....	29
2.2.33. ssyr2k.....	29
2.2.34. ssyrkx.....	30
2.2.35. strmm.....	30
2.2.36. strsm.....	31
2.2.37. cublasSgemvBatched.....	31
2.2.38. cublasSgemmBatched.....	32
2.2.39. cublasSgelsBatched.....	32
2.2.40. cublasSgeqrfBatched.....	33
2.2.41. cublasSgetrfBatched.....	33
2.2.42. cublasSgetriBatched.....	33
2.2.43. cublasSgetrsBatched.....	34

2.2.44. cublasSmatinvBatched.....	34
2.2.45. cublasStrsmBatched.....	34
2.2.46. cublasSgemvStridedBatched.....	35
2.2.47. cublasSgemmStridedBatched.....	35
2.3. Double Precision Functions and Subroutines.....	36
2.3.1. idamax.....	36
2.3.2. idamin.....	37
2.3.3. dasum.....	37
2.3.4. daxpy.....	37
2.3.5. dcopy.....	38
2.3.6. ddot.....	38
2.3.7. dnorm2.....	38
2.3.8. drot.....	39
2.3.9. drotg.....	39
2.3.10. drotm.....	39
2.3.11. drotmg.....	40
2.3.12. dscal.....	40
2.3.13. dswap.....	40
2.3.14. dgbmv.....	41
2.3.15. dgemv.....	41
2.3.16. dger.....	42
2.3.17. dsbmv.....	42
2.3.18. dspmv.....	42
2.3.19. dspr.....	43
2.3.20. dspr2.....	43
2.3.21. dsymv.....	44
2.3.22. dsyr.....	44
2.3.23. dsyr2.....	44
2.3.24. dtbmv.....	45
2.3.25. dtbsv.....	45
2.3.26. dtpmv.....	46
2.3.27. dtpsv.....	46
2.3.28. dtrmv.....	46
2.3.29. dtrsv.....	47
2.3.30. dgemm.....	47
2.3.31. dsymm.....	48
2.3.32. dsyrk.....	48
2.3.33. dsyr2k.....	49
2.3.34. dsyrkx.....	49
2.3.35. dtrmm.....	50
2.3.36. dtrsm.....	50
2.3.37. cublasDgemvBatched.....	51
2.3.38. cublasDgemmBatched.....	51

2.3.39. cublasDgelsBatched.....	52
2.3.40. cublasDgeqrfBatched.....	52
2.3.41. cublasDgetrfBatched.....	53
2.3.42. cublasDgetriBatched.....	53
2.3.43. cublasDgetrsBatched.....	53
2.3.44. cublasDmatinvBatched.....	54
2.3.45. cublasDtrsmBatched.....	54
2.3.46. cublasDgemvStridedBatched.....	54
2.3.47. cublasDgemmStridedBatched.....	55
2.4. Single Precision Complex Functions and Subroutines.....	56
2.4.1. icamax.....	56
2.4.2. icamin.....	56
2.4.3. scasum.....	57
2.4.4. caxpy.....	57
2.4.5. ccopy.....	57
2.4.6. cdotc.....	58
2.4.7. cdotu.....	58
2.4.8. scnrm2.....	58
2.4.9. crot.....	59
2.4.10. csrot.....	59
2.4.11. crotg.....	59
2.4.12. cscal.....	60
2.4.13. csscal.....	60
2.4.14. cswap.....	60
2.4.15. cgbmv.....	61
2.4.16. cgemv.....	61
2.4.17. cgerc.....	62
2.4.18. cgeru.....	62
2.4.19. csymv.....	62
2.4.20. csyr.....	63
2.4.21. csyr2.....	63
2.4.22. ctbmv.....	64
2.4.23. ctbsv.....	64
2.4.24. ctpmv.....	64
2.4.25. ctpsv.....	65
2.4.26. ctrmv.....	65
2.4.27. ctrsv.....	66
2.4.28. chbmv.....	66
2.4.29. chemv.....	66
2.4.30. chpmv.....	67
2.4.31. cher.....	67
2.4.32. cher2.....	68
2.4.33. chpr.....	68

2.4.34. chpr2.....	68
2.4.35. cgemm.....	69
2.4.36. csymm.....	69
2.4.37. csyrk.....	70
2.4.38. csyr2k.....	70
2.4.39. csyrkx.....	71
2.4.40. ctrmm.....	71
2.4.41. ctrsm.....	72
2.4.42. chemm.....	72
2.4.43. cherk.....	73
2.4.44. cher2k.....	73
2.4.45. cherkx.....	74
2.4.46. cublasCgemvBatched.....	74
2.4.47. cublasCgemmBatched.....	75
2.4.48. cublasCgelsBatched.....	75
2.4.49. cublasCgeqrfBatched.....	76
2.4.50. cublasCgetrfBatched.....	76
2.4.51. cublasCgetriBatched.....	77
2.4.52. cublasCgetrsBatched.....	77
2.4.53. cublasCmatinvBatched.....	77
2.4.54. cublasCtrsmBatched.....	77
2.4.55. cublasCgemvStridedBatched.....	78
2.4.56. cublasCgemmStridedBatched.....	79
2.5. Double Precision Complex Functions and Subroutines.....	79
2.5.1. izamax.....	80
2.5.2. izamin.....	80
2.5.3. dzasum.....	80
2.5.4. zaxpy.....	81
2.5.5. zcopy.....	81
2.5.6. zdotc.....	81
2.5.7. zdotu.....	82
2.5.8. dznrm2.....	82
2.5.9. zrot.....	82
2.5.10. zsrot.....	83
2.5.11. zrotg.....	83
2.5.12. zscal.....	83
2.5.13. zdscal.....	84
2.5.14. zswap.....	84
2.5.15. zgbmv.....	84
2.5.16. zgemv.....	85
2.5.17. zgerc.....	85
2.5.18. zgeru.....	86
2.5.19. zsymv.....	86

2.5.20. zsyr.....	86
2.5.21. zsyr2.....	87
2.5.22. ztbmv.....	87
2.5.23. ztbsv.....	88
2.5.24. ztpmv.....	88
2.5.25. ztpsv.....	88
2.5.26. ztrmv.....	89
2.5.27. ztrsv.....	89
2.5.28. zhbmv.....	90
2.5.29. zhemv.....	90
2.5.30. zhpmv.....	91
2.5.31. zher.....	91
2.5.32. zher2.....	91
2.5.33. zhpr.....	92
2.5.34. zhpr2.....	92
2.5.35. zgemm.....	93
2.5.36. zsymm.....	93
2.5.37. zsyrk.....	94
2.5.38. zsyr2k.....	94
2.5.39. zsyrkx.....	95
2.5.40. ztrmm.....	95
2.5.41. ztrsm.....	96
2.5.42. zhemm.....	96
2.5.43. zherk.....	97
2.5.44. zher2k.....	97
2.5.45. zherkx.....	98
2.5.46. cublasZgemvBatched.....	98
2.5.47. cublasZgemmBatched.....	99
2.5.48. cublasZgelsBatched.....	99
2.5.49. cublasZgeqrfBatched.....	100
2.5.50. cublasZgetrfBatched.....	100
2.5.51. cublasZgetriBatched.....	100
2.5.52. cublasZgetrsBatched.....	101
2.5.53. cublasZmatinvBatched.....	101
2.5.54. cublasZtrsmBatched.....	101
2.5.55. cublasZgemvStridedBatched.....	102
2.5.56. cublasZgemmStridedBatched.....	102
2.6. Half Precision Functions and Extension Functions.....	103
2.6.1. cublasHgemvBatched.....	104
2.6.2. cublasHgemvStridedBatched.....	105
2.6.3. cublasHgemm.....	106
2.6.4. cublasHgemmBatched.....	106
2.6.5. cublasHgemmStridedBatched.....	107

2.6.6. cublaslamaxEx.....	108
2.6.7. cublaslaminEx.....	108
2.6.8. cublasAsumEx.....	108
2.6.9. cublasAxyEx.....	108
2.6.10. cublasCopyEx.....	109
2.6.11. cublasDotEx.....	109
2.6.12. cublasDotcEx.....	109
2.6.13. cublasNrm2Ex.....	109
2.6.14. cublasRotEx.....	110
2.6.15. cublasRotgEx.....	110
2.6.16. cublasRotmEx.....	110
2.6.17. cublasRotmgEx.....	110
2.6.18. cublasScalEx.....	111
2.6.19. cublasSwapEx.....	111
2.6.20. cublasGemmEx.....	111
2.6.21. cublasGemmBatchedEx.....	112
2.6.22. cublasGemmStridedBatchedEx.....	112
2.7. CUBLAS V2 Module Functions.....	113
2.7.1. Single Precision Functions and Subroutines.....	113
2.7.1.1. isamax.....	113
2.7.1.2. isamin.....	113
2.7.1.3. sasum.....	113
2.7.1.4. saxpy.....	114
2.7.1.5. scopy.....	114
2.7.1.6. sdot.....	114
2.7.1.7. snrm2.....	114
2.7.1.8. srot.....	114
2.7.1.9. srotg.....	115
2.7.1.10. srotm.....	115
2.7.1.11. srotmg.....	115
2.7.1.12. sscal.....	115
2.7.1.13. sswap.....	115
2.7.1.14. sgbmv.....	115
2.7.1.15. sgemv.....	116
2.7.1.16. sger.....	116
2.7.1.17. ssbmv.....	116
2.7.1.18. sspmv.....	116
2.7.1.19. sspr.....	116
2.7.1.20. sspr2.....	117
2.7.1.21. ssymv.....	117
2.7.1.22. ssyr.....	117
2.7.1.23. ssyr2.....	117
2.7.1.24. stbmv.....	117



2.7.1.25. stbsv.....	118
2.7.1.26. stpmv.....	118
2.7.1.27. stpsv.....	118
2.7.1.28. strmv.....	118
2.7.1.29. strsv.....	118
2.7.1.30. sgemm.....	119
2.7.1.31. ssymm.....	119
2.7.1.32. ssyrk.....	119
2.7.1.33. ssyr2k.....	119
2.7.1.34. ssyrkx.....	120
2.7.1.35. strmm.....	120
2.7.1.36. strsm.....	120
2.7.2. Double Precision Functions and Subroutines.....	120
2.7.2.1. idamax.....	120
2.7.2.2. idamin.....	121
2.7.2.3. dasum.....	121
2.7.2.4. daxpy.....	121
2.7.2.5. dcopy.....	121
2.7.2.6. ddot.....	121
2.7.2.7. dnorm2.....	122
2.7.2.8. drot.....	122
2.7.2.9. drotg.....	122
2.7.2.10. drotm.....	122
2.7.2.11. drotmg.....	122
2.7.2.12. dscal.....	122
2.7.2.13. dswap.....	123
2.7.2.14. dgbmv.....	123
2.7.2.15. dgemv.....	123
2.7.2.16. dger.....	123
2.7.2.17. dsbmv.....	123
2.7.2.18. dspmv.....	124
2.7.2.19. dspr.....	124
2.7.2.20. dspr2.....	124
2.7.2.21. dsymv.....	124
2.7.2.22. dsyr.....	124
2.7.2.23. dsyr2.....	125
2.7.2.24. dtbmv.....	125
2.7.2.25. dtbsv.....	125
2.7.2.26. dtpmv.....	125
2.7.2.27. dtpsv.....	125
2.7.2.28. dtrmv.....	126
2.7.2.29. dtrsv.....	126
2.7.2.30. dgemm.....	126

2.7.2.31. dsymm.....	126
2.7.2.32. dsyrk.....	127
2.7.2.33. dsyr2k.....	127
2.7.2.34. dsyrkx.....	127
2.7.2.35. dtrmm.....	127
2.7.2.36. dtrsm.....	128
2.7.3. Single Precision Complex Functions and Subroutines.....	128
2.7.3.1. icamax.....	128
2.7.3.2. icamin.....	128
2.7.3.3. scasum.....	128
2.7.3.4. caxpy.....	129
2.7.3.5. ccopy.....	129
2.7.3.6. cdotc.....	129
2.7.3.7. cdotu.....	129
2.7.3.8. scnrm2.....	129
2.7.3.9. crot.....	130
2.7.3.10. csrot.....	130
2.7.3.11. crotg.....	130
2.7.3.12. cscal.....	130
2.7.3.13. csscal.....	130
2.7.3.14. cswap.....	131
2.7.3.15. cgmv.....	131
2.7.3.16. cgemv.....	131
2.7.3.17. cgerc.....	131
2.7.3.18. cgeru.....	131
2.7.3.19. csymv.....	132
2.7.3.20. csyr.....	132
2.7.3.21. csyr2.....	132
2.7.3.22. ctbmv.....	132
2.7.3.23. ctbsv.....	132
2.7.3.24. ctpmv.....	133
2.7.3.25. ctpsv.....	133
2.7.3.26. ctrmv.....	133
2.7.3.27. ctrsv.....	133
2.7.3.28. chbmv.....	133
2.7.3.29. chemv.....	134
2.7.3.30. chpmv.....	134
2.7.3.31. cher.....	134
2.7.3.32. cher2.....	134
2.7.3.33. chpr.....	134
2.7.3.34. chpr2.....	135
2.7.3.35. cgemm.....	135
2.7.3.36. csymm.....	135

2.7.3.37. csyrk.....	135
2.7.3.38. csyr2k.....	136
2.7.3.39. csyrkx.....	136
2.7.3.40. ctrmm.....	136
2.7.3.41. ctrsm.....	136
2.7.3.42. chemm.....	137
2.7.3.43. cherk.....	137
2.7.3.44. cher2k.....	137
2.7.3.45. cherkx.....	137
2.7.4. Double Precision Complex Functions and Subroutines.....	138
2.7.4.1. izamax.....	138
2.7.4.2. izamin.....	138
2.7.4.3. dzasum.....	138
2.7.4.4. zaxpy.....	138
2.7.4.5. zcopy.....	139
2.7.4.6. zdotc.....	139
2.7.4.7. zdotu.....	139
2.7.4.8. dznrm2.....	139
2.7.4.9. zrot.....	139
2.7.4.10. zsrot.....	140
2.7.4.11. zrotg.....	140
2.7.4.12. zscal.....	140
2.7.4.13. zdscal.....	140
2.7.4.14. zswap.....	140
2.7.4.15. zgbmv.....	141
2.7.4.16. zgemv.....	141
2.7.4.17. zgerc.....	141
2.7.4.18. zgeru.....	141
2.7.4.19. zsymv.....	141
2.7.4.20. zsyr.....	142
2.7.4.21. zsyr2.....	142
2.7.4.22. ztbmv.....	142
2.7.4.23. ztbsv.....	142
2.7.4.24. ztpmv.....	142
2.7.4.25. ztpsv.....	143
2.7.4.26. ztrmv.....	143
2.7.4.27. ztrsv.....	143
2.7.4.28. zhbmv.....	143
2.7.4.29. zhemv.....	143
2.7.4.30. zhpmv.....	144
2.7.4.31. zher.....	144
2.7.4.32. zher2.....	144
2.7.4.33. zhpr.....	144

2.7.4.34. zhpr2.....	144
2.7.4.35. zgemm.....	145
2.7.4.36. zsymm.....	145
2.7.4.37. zsyrk.....	145
2.7.4.38. zsyr2k.....	145
2.7.4.39. zsyrkx.....	146
2.7.4.40. ztrmm.....	146
2.7.4.41. ztrsm.....	146
2.7.4.42. zhemm.....	146
2.7.4.43. zherk.....	147
2.7.4.44. zher2k.....	147
2.7.4.45. zherkx.....	147
2.8. CUBLAS XT Module Functions.....	147
2.8.1. cublasXtCreate.....	148
2.8.2. cublasXtDestroy.....	149
2.8.3. cublasXtDeviceSelect.....	149
2.8.4. cublasXtSetBlockDim.....	149
2.8.5. cublasXtGetBlockDim.....	149
2.8.6. cublasXtSetCpuRoutine.....	149
2.8.7. cublasXtSetCpuRatio.....	150
2.8.8. cublasXtSetPinningMemMode.....	150
2.8.9. cublasXtGetPinningMemMode.....	150
2.8.10. cublasXtSgemm.....	150
2.8.11. cublasXtSsymm.....	151
2.8.12. cublasXtSsyrk.....	151
2.8.13. cublasXtSsyr2k.....	151
2.8.14. cublasXtSsyrkx.....	151
2.8.15. cublasXtStrmm.....	152
2.8.16. cublasXtStrsm.....	152
2.8.17. cublasXtSspmm.....	152
2.8.18. cublasXtCgemm.....	152
2.8.19. cublasXtChemmm.....	153
2.8.20. cublasXtCherk.....	153
2.8.21. cublasXtCher2k.....	153
2.8.22. cublasXtCherkx.....	154
2.8.23. cublasXtCsymm.....	154
2.8.24. cublasXtCsyrk.....	154
2.8.25. cublasXtCsyr2k.....	154
2.8.26. cublasXtCsyrkx.....	155
2.8.27. cublasXtCtrmm.....	155
2.8.28. cublasXtCtrsm.....	155
2.8.29. cublasXtCspmm.....	156
2.8.30. cublasXtDgemm.....	156

2.8.31. cublasXtDsymb.....	156
2.8.32. cublasXtDsyrk.....	156
2.8.33. cublasXtDsyr2k.....	157
2.8.34. cublasXtDsyrkx.....	157
2.8.35. cublasXtDtrmm.....	157
2.8.36. cublasXtDtrsm.....	157
2.8.37. cublasXtDspmm.....	158
2.8.38. cublasXtZgemm.....	158
2.8.39. cublasXtZhemm.....	158
2.8.40. cublasXtZherk.....	159
2.8.41. cublasXtZher2k.....	159
2.8.42. cublasXtZherkx.....	159
2.8.43. cublasXtZsymm.....	160
2.8.44. cublasXtZsyrk.....	160
2.8.45. cublasXtZsyr2k.....	160
2.8.46. cublasXtZsyrkx.....	160
2.8.47. cublasXtZtrmm.....	161
2.8.48. cublasXtZtrsm.....	161
2.8.49. cublasXtZspmm.....	161
Chapter 3. FFT Runtime Library APIs.....	162
3.1. CUFFT Definitions and Helper Functions.....	162
3.1.1. cufftSetCompatibilityMode.....	163
3.1.2. cufftSetStream.....	163
3.1.3. cufftGetVersion.....	163
3.1.4. cufftSetAutoAllocation.....	163
3.1.5. cufftSetWorkArea.....	164
3.1.6. cufftDestroy.....	164
3.2. CUFFT Plans and Estimated Size Functions.....	164
3.2.1. cufftPlan1d.....	164
3.2.2. cufftPlan2d.....	164
3.2.3. cufftPlan3d.....	164
3.2.4. cufftPlanMany.....	165
3.2.5. cufftCreate.....	165
3.2.6. cufftMakePlan1d.....	165
3.2.7. cufftMakePlan2d.....	165
3.2.8. cufftMakePlan3d.....	166
3.2.9. cufftMakePlanMany.....	166
3.2.10. cufftEstimate1d.....	166
3.2.11. cufftEstimate2d.....	167
3.2.12. cufftEstimate3d.....	167
3.2.13. cufftEstimateMany.....	167
3.2.14. cufftGetSize1d.....	167
3.2.15. cufftGetSize2d.....	167

3.2.16. cufftGetSize3d.....	168
3.2.17. cufftGetSizeMany.....	168
3.2.18. cufftGetSize.....	168
3.3. CUFFT Execution Functions.....	168
3.3.1. cufftExecC2C.....	168
3.3.2. cufftExecR2C.....	169
3.3.3. cufftExecC2R.....	169
3.3.4. cufftExecZ2Z.....	169
3.3.5. cufftExecD2Z.....	169
3.3.6. cufftExecZ2D.....	169
3.4. CUFFTXT Definitions and Helper Functions.....	170
3.4.1. cufftXtSetGPUs.....	171
3.4.2. cufftXtMalloc.....	172
3.4.3. cufftXtFree.....	172
3.4.4. cufftXtMemcpy.....	172
3.5. CUFFTXT Plans and Work Area Functions.....	172
3.5.1. cufftXtMakePlanMany.....	172
3.5.2. cufftXtQueryPlan.....	173
3.5.3. cufftXtSetWorkAreaPolicy.....	173
3.5.4. cufftXtGetSizeMany.....	173
3.5.5. cufftXtSetWorkArea.....	174
3.5.6. cufftXtSetDistribution.....	174
3.6. CUFFTXT Execution Functions.....	174
3.6.1. cufftXtExec.....	174
3.6.2. cufftXtExecDescriptor.....	174
3.6.3. cufftXtExecDescriptorC2C.....	175
3.6.4. cufftXtExecDescriptorZ2Z.....	175
3.6.5. cufftXtExecDescriptorR2C.....	175
3.6.6. cufftXtExecDescriptorD2Z.....	175
3.6.7. cufftXtExecDescriptorC2R.....	175
3.6.8. cufftXtExecDescriptorZ2D.....	176
3.7. CUFFTMP Functions.....	176
3.7.1. cufftMpNvshmemMalloc.....	176
3.7.2. cufftMpNvshmemFree.....	176
3.7.3. cufftMpAttachComm.....	176
3.7.4. cufftMpCreateReshape.....	177
3.7.5. cufftMpAttachReshapeComm.....	177
3.7.6. cufftMpGetReshapeSize.....	177
3.7.7. cufftMpMakeReshape.....	177
3.7.8. cufftMpExecReshapeAsync.....	177
3.7.9. cufftMpDestroyReshape.....	177
Chapter 4. Random Number Runtime Library APIs.....	178
4.1. CURAND Definitions and Helper Functions.....	178

4.1.1. curandCreateGenerator.....	180
4.1.2. curandCreateGeneratorHost.....	180
4.1.3. curandDestroyGenerator.....	180
4.1.4. curandGetVersion.....	180
4.1.5. curandSetStream.....	180
4.1.6. curandSetPseudoRandomGeneratorSeed.....	180
4.1.7. curandSetGeneratorOffset.....	180
4.1.8. curandSetGeneratorOrdering.....	181
4.1.9. curandSetQuasiRandomGeneratorDimensions.....	181
4.2. CURAND Generator Functions.....	181
4.2.1. curandGenerate.....	181
4.2.2. curandGenerateLongLong.....	181
4.2.3. curandGenerateUniform.....	181
4.2.4. curandGenerateUniformDouble.....	182
4.2.5. curandGenerateNormal.....	182
4.2.6. curandGenerateNormalDouble.....	182
4.2.7. curandGeneratePoisson.....	182
4.2.8. curandGenerateSeeds.....	182
4.2.9. curandGenerateLogNormal.....	182
4.2.10. curandGenerateLogNormalDouble.....	183
4.3. CURAND Device Definitions and Functions.....	183
4.3.1. curand_Init.....	184
4.3.1.1. curandInitXORWOW.....	184
4.3.1.2. curandInitMRG32k3a.....	184
4.3.1.3. curandInitPhilox4_32_10.....	184
4.3.1.4. curandInitSobol32.....	185
4.3.1.5. curandInitScrambledSobol32.....	185
4.3.1.6. curandInitSobol64.....	185
4.3.1.7. curandInitScrambledSobol64.....	185
4.3.2. curand.....	186
4.3.2.1. curandGetXORWOW.....	186
4.3.2.2. curandGetMRG32k3a.....	186
4.3.2.3. curandGetPhilox4_32_10.....	186
4.3.2.4. curandGetSobol32.....	186
4.3.2.5. curandGetScrambledSobol32.....	186
4.3.2.6. curandGetSobol64.....	187
4.3.2.7. curandGetScrambledSobol64.....	187
4.3.3. Curand_Normal.....	187
4.3.3.1. curandNormalXORWOW.....	187
4.3.3.2. curandNormalMRG32k3a.....	187
4.3.3.3. curandNormalPhilox4_32_10.....	187
4.3.3.4. curandNormalSobol32.....	188
4.3.3.5. curandNormalScrambledSobol32.....	188

4.3.3.6. curandNormalSobol64.....	188
4.3.3.7. curandNormalScrambledSobol64.....	188
4.3.4. Curand_Normal_Double.....	188
4.3.4.1. curandNormalDoubleXORWOW.....	188
4.3.4.2. curandNormalDoubleMRG32k3a.....	189
4.3.4.3. curandNormalDoublePhilox4_32_10.....	189
4.3.4.4. curandNormalDoubleSobol32.....	189
4.3.4.5. curandNormalDoubleScrambledSobol32.....	189
4.3.4.6. curandNormalDoubleSobol64.....	189
4.3.4.7. curandNormalDoubleScrambledSobol64.....	189
4.3.5. Curand_Log_Normal.....	190
4.3.5.1. curandLogNormalXORWOW.....	190
4.3.5.2. curandLogNormalMRG32k3a.....	190
4.3.5.3. curandLogNormalPhilox4_32_10.....	190
4.3.5.4. curandLogNormalSobol32.....	190
4.3.5.5. curandLogNormalScrambledSobol32.....	190
4.3.5.6. curandLogNormalSobol64.....	191
4.3.5.7. curandLogNormalScrambledSobol64.....	191
4.3.6. Curand_Log_Normal_Double.....	191
4.3.6.1. curandLogNormalDoubleXORWOW.....	191
4.3.6.2. curandLogNormalDoubleMRG32k3a.....	191
4.3.6.3. curandLogNormalDoublePhilox4_32_10.....	191
4.3.6.4. curandLogNormalDoubleSobol32.....	192
4.3.6.5. curandLogNormalDoubleScrambledSobol32.....	192
4.3.6.6. curandLogNormalDoubleSobol64.....	192
4.3.6.7. curandLogNormalDoubleScrambledSobol64.....	192
4.3.7. Curand_Uniform.....	192
4.3.7.1. curandUniformXORWOW.....	193
4.3.7.2. curandUniformMRG32k3a.....	193
4.3.7.3. curandUniformPhilox4_32_10.....	193
4.3.7.4. curandUniformSobol32.....	193
4.3.7.5. curandUniformScrambledSobol32.....	193
4.3.7.6. curandUniformSobol64.....	193
4.3.7.7. curandUniformScrambledSobol64.....	194
4.3.8. Curand_Uniform_Double.....	194
4.3.8.1. curandUniformDoubleXORWOW.....	194
4.3.8.2. curandUniformDoubleMRG32k3a.....	194
4.3.8.3. curandUniformDoublePhilox4_32_10.....	194
4.3.8.4. curandUniformDoubleSobol32.....	194
4.3.8.5. curandUniformDoubleScrambledSobol32.....	195
4.3.8.6. curandUniformDoubleSobol64.....	195
4.3.8.7. curandUniformDoubleScrambledSobol64.....	195
Chapter 5. SPARSE Matrix Runtime Library APIs.....	196



5.1. CUSPARSE Definitions and Helper Functions.....	196
5.1.1. cusparseCreate.....	200
5.1.2. cusparseDestroy.....	200
5.1.3. cusparseGetErrorName.....	200
5.1.4. cusparseGetErrorString.....	200
5.1.5. cusparseGetVersion.....	201
5.1.6. cusparseSetStream.....	201
5.1.7. cusparseGetStream.....	201
5.1.8. cusparseGetPointerMode.....	201
5.1.9. cusparseSetPointerMode.....	201
5.1.10. cusparseCreateMatDescr.....	201
5.1.11. cusparseDestroyMatDescr.....	202
5.1.12. cusparseSetMatType.....	202
5.1.13. cusparseGetMatType.....	202
5.1.14. cusparseSetMatFillMode.....	202
5.1.15. cusparseGetMatFillMode.....	202
5.1.16. cusparseSetMatDiagType.....	202
5.1.17. cusparseGetMatDiagType.....	202
5.1.18. cusparseSetMatIndexBase.....	202
5.1.19. cusparseGetMatIndexBase.....	203
5.1.20. cusparseCreateSolveAnalysisInfo.....	203
5.1.21. cusparseDestroySolveAnalysisInfo.....	203
5.1.22. cusparseGetLevelInfo.....	203
5.1.23. cusparseCreateHybMat.....	203
5.1.24. cusparseDestroyHybMat.....	203
5.1.25. cusparseCreateCsrsv2Info.....	204
5.1.26. cusparseDestroyCsrsv2Info.....	204
5.1.27. cusparseCreateCsr02Info.....	204
5.1.28. cusparseDestroyCsr02Info.....	204
5.1.29. cusparseCreateCsrilu02Info.....	204
5.1.30. cusparseDestroyCsrilu02Info.....	204
5.1.31. cusparseCreateBsrsv2Info.....	204
5.1.32. cusparseDestroyBsrsv2Info.....	204
5.1.33. cusparseCreateBsr02Info.....	205
5.1.34. cusparseDestroyBsr02Info.....	205
5.1.35. cusparseCreateBsrilu02Info.....	205
5.1.36. cusparseDestroyBsrilu02Info.....	205
5.1.37. cusparseCreateBsrsm2Info.....	205
5.1.38. cusparseDestroyBsrsm2Info.....	205
5.1.39. cusparseCreateCsrgemm2Info.....	205
5.1.40. cusparseDestroyCsrgemm2Info.....	206
5.1.41. cusparseCreateColorInfo.....	206
5.1.42. cusparseDestroyColorInfo.....	206

5.1.43. cusparseCreateCsr2csrInfo.....	206
5.1.44. cusparseDestroyCsr2csrInfo.....	206
5.2. CUSPARSE Level 1 Functions.....	206
5.2.1. cusparseSaxpyi.....	206
5.2.2. cusparseDaxpyi.....	207
5.2.3. cusparseCaxpyi.....	207
5.2.4. cusparseZaxpyi.....	207
5.2.5. cusparseSdoti.....	207
5.2.6. cusparseDdoti.....	208
5.2.7. cusparseCdoti.....	208
5.2.8. cusparseZdoti.....	208
5.2.9. cusparseCdotci.....	208
5.2.10. cusparseZdotci.....	209
5.2.11. cusparseSgthr.....	209
5.2.12. cusparseDgthr.....	209
5.2.13. cusparseCgthr.....	209
5.2.14. cusparseZgthr.....	210
5.2.15. cusparseSgthrz.....	210
5.2.16. cusparseDgthrz.....	210
5.2.17. cusparseCgthrz.....	210
5.2.18. cusparseZgthrz.....	211
5.2.19. cusparseSsctr.....	211
5.2.20. cusparseDsctr.....	211
5.2.21. cusparseCsctr.....	211
5.2.22. cusparseZsctr.....	211
5.2.23. cusparseSroti.....	212
5.2.24. cusparseDroti.....	212
5.3. CUSPARSE Level 2 Functions.....	212
5.3.1. cusparseSbsrmv.....	212
5.3.2. cusparseDbsrmv.....	213
5.3.3. cusparseCbsrmv.....	213
5.3.4. cusparseZbsrmv.....	213
5.3.5. cusparseSbsrxmv.....	214
5.3.6. cusparseDbsrxmv.....	214
5.3.7. cusparseCbsrxmv.....	214
5.3.8. cusparseZbsrxmv.....	215
5.3.9. cusparseScsrmv.....	215
5.3.10. cusparseDcsrmv.....	215
5.3.11. cusparseCcsrmv.....	216
5.3.12. cusparseZcsrmv.....	216
5.3.13. cusparseScsrsv_analysis.....	217
5.3.14. cusparseDcsrsv_analysis.....	217
5.3.15. cusparseCcsrsv_analysis.....	217

5.3.16. cusparseZcsrsv_analysis.....	217
5.3.17. cusparseScsrsv_solve.....	218
5.3.18. cusparseDcsrsv_solve.....	218
5.3.19. cusparseCcsrsv_solve.....	218
5.3.20. cusparseZcsrsv_solve.....	218
5.3.21. cusparseSgemvi_bufferSize.....	219
5.3.22. cusparseDgemvi_bufferSize.....	219
5.3.23. cusparseCgemvi_bufferSize.....	219
5.3.24. cusparseZgemvi_bufferSize.....	219
5.3.25. cusparseSgemvi.....	219
5.3.26. cusparseDgemvi.....	220
5.3.27. cusparseCgemvi.....	220
5.3.28. cusparseZgemvi.....	220
5.3.29. cusparseShybm.....	221
5.3.30. cusparseDhybm.....	221
5.3.31. cusparseChybm.....	221
5.3.32. cusparseZhybm.....	222
5.3.33. cusparseShybsv_analysis.....	222
5.3.34. cusparseDhybsv_analysis.....	222
5.3.35. cusparseChybsv_analysis.....	222
5.3.36. cusparseZhybsv_analysis.....	222
5.3.37. cusparseShybsv_solve.....	223
5.3.38. cusparseDhybsv_solve.....	223
5.3.39. cusparseChybsv_solve.....	223
5.3.40. cusparseZhybsv_solve.....	223
5.3.41. cusparseSbsrsv2_bufferSize.....	224
5.3.42. cusparseDbbsrsv2_bufferSize.....	224
5.3.43. cusparseCbbsrsv2_bufferSize.....	224
5.3.44. cusparseZbbsrsv2_bufferSize.....	224
5.3.45. cusparseSbsrsv2_analysis.....	225
5.3.46. cusparseDbbsrsv2_analysis.....	225
5.3.47. cusparseCbbsrsv2_analysis.....	225
5.3.48. cusparseZbbsrsv2_analysis.....	225
5.3.49. cusparseSbsrsv2_solve.....	226
5.3.50. cusparseDbbsrsv2_solve.....	226
5.3.51. cusparseCbbsrsv2_solve.....	226
5.3.52. cusparseZbbsrsv2_solve.....	227
5.3.53. cusparseXbsrsv2_zeroPivot.....	227
5.3.54. cusparseScsrsv2_bufferSize.....	227
5.3.55. cusparseDcsrsv2_bufferSize.....	227
5.3.56. cusparseCcsrsv2_bufferSize.....	228
5.3.57. cusparseZcsrsv2_bufferSize.....	228
5.3.58. cusparseScsrsv2_analysis.....	228

5.3.59. cusparseDcsrsv2_analysis.....	228
5.3.60. cusparseCcsrsv2_analysis.....	229
5.3.61. cusparseZcsrsv2_analysis.....	229
5.3.62. cusparseScsrsv2_solve.....	229
5.3.63. cusparseDcsrsv2_solve.....	229
5.3.64. cusparseCcsrsv2_solve.....	230
5.3.65. cusparseZcsrsv2_solve.....	230
5.3.66. cusparseXcsrsv2_zeroPivot.....	230
5.4. CUSPARSE Level 3 Functions.....	230
5.4.1. cusparseScsrmm.....	230
5.4.2. cusparseDcsrmm.....	231
5.4.3. cusparseCcsrmm.....	231
5.4.4. cusparseZcsrmm.....	231
5.4.5. cusparseScsrmm2.....	232
5.4.6. cusparseDcsrmm2.....	232
5.4.7. cusparseCcsrmm2.....	232
5.4.8. cusparseZcsrmm2.....	233
5.4.9. cusparseScsrsm_analysis.....	233
5.4.10. cusparseDcsrsm_analysis.....	233
5.4.11. cusparseCcsrsm_analysis.....	234
5.4.12. cusparseZcsrsm_analysis.....	234
5.4.13. cusparseScsrsm_solve.....	234
5.4.14. cusparseDcsrsm_solve.....	234
5.4.15. cusparseCcsrsm_solve.....	235
5.4.16. cusparseZcsrsm_solve.....	235
5.4.17. cusparseScsrsm2_bufferSizeExt.....	235
5.4.18. cusparseDcsrsm2_bufferSizeExt.....	235
5.4.19. cusparseCcsrsm2_bufferSizeExt.....	236
5.4.20. cusparseZcsrsm2_bufferSizeExt.....	236
5.4.21. cusparseScsrsm2_analysis.....	236
5.4.22. cusparseDcsrsm2_analysis.....	236
5.4.23. cusparseCcsrsm2_analysis.....	237
5.4.24. cusparseZcsrsm2_analysis.....	237
5.4.25. cusparseScsrsm2_solve.....	237
5.4.26. cusparseDcsrsm2_solve.....	238
5.4.27. cusparseCcsrsm2_solve.....	238
5.4.28. cusparseZcsrsm2_solve.....	238
5.4.29. cusparseXcsrsm2_zeroPivot.....	239
5.4.30. cusparseSbsrmm.....	239
5.4.31. cusparseDbrrmm.....	239
5.4.32. cusparseCbsrmm.....	239
5.4.33. cusparseZbsrmm.....	240
5.4.34. cusparseSbsrsm2_bufferSize.....	240

5.4.35. cusparseDbsrsm2_bufferSize.....	240
5.4.36. cusparseCbsrsm2_bufferSize.....	240
5.4.37. cusparseZbsrsm2_bufferSize.....	241
5.4.38. cusparseSbsrsm2_analysis.....	241
5.4.39. cusparseDbsrsm2_analysis.....	241
5.4.40. cusparseCbsrsm2_analysis.....	241
5.4.41. cusparseZbsrsm2_analysis.....	242
5.4.42. cusparseSbsrsm2_solve.....	242
5.4.43. cusparseDbsrsm2_solve.....	242
5.4.44. cusparseCbsrsm2_solve.....	242
5.4.45. cusparseZbsrsm2_solve.....	243
5.4.46. cusparseXbsrsm2_zeroPivot.....	243
5.4.47. cusparseSgemmi.....	243
5.4.48. cusparseDgemmi.....	243
5.4.49. cusparseCgemmi.....	244
5.4.50. cusparseZgemmi.....	244
5.5. CUSPARSE Extra Functions.....	245
5.5.1. cusparseXcsrgeamNnz.....	245
5.5.2. cusparseScsrgeam.....	245
5.5.3. cusparseDcsrgeam.....	245
5.5.4. cusparseCcsrgeam.....	246
5.5.5. cusparseZcsrgeam.....	246
5.5.6. cusparseScsrgeam2_bufferSizeExt.....	247
5.5.7. cusparseDcsrgeam2_bufferSizeExt.....	247
5.5.8. cusparseCcsrgeam2_bufferSizeExt.....	247
5.5.9. cusparseZcsrgeam2_bufferSizeExt.....	248
5.5.10. cusparseXcsrgeam2Nnz.....	248
5.5.11. cusparseScsrgeam2.....	248
5.5.12. cusparseDcsrgeam2.....	249
5.5.13. cusparseCcsrgeam2.....	249
5.5.14. cusparseZcsrgeam2.....	249
5.5.15. cusparseXcsrgeammNnz.....	250
5.5.16. cusparseScsrgeamm.....	250
5.5.17. cusparseDcsrgeamm.....	250
5.5.18. cusparseCcsrgeamm.....	251
5.5.19. cusparseZcsrgeamm.....	251
5.5.20. cusparseScsrgeamm2_bufferSizeExt.....	251
5.5.21. cusparseDcsrgeamm2_bufferSizeExt.....	252
5.5.22. cusparseCcsrgeamm2_bufferSizeExt.....	252
5.5.23. cusparseZcsrgeamm2_bufferSizeExt.....	252
5.5.24. cusparseXcsrgeamm2Nnz.....	252
5.5.25. cusparseScsrgeamm2.....	253
5.5.26. cusparseDcsrgeamm2.....	253

5.5.27. cusparseCcsrgemm2.....	253
5.5.28. cusparseZcsrgemm2.....	254
5.6. CUSPARSE Preconditioning Functions.....	254
5.6.1. cusparseScsric0.....	254
5.6.2. cusparseDcsric0.....	255
5.6.3. cusparseCcsric0.....	255
5.6.4. cusparseZcsric0.....	255
5.6.5. cusparseScsrilu0.....	256
5.6.6. cusparseDcsrilu0.....	256
5.6.7. cusparseCcsrilu0.....	256
5.6.8. cusparseZcsrilu0.....	256
5.6.9. cusparseSgtsv.....	257
5.6.10. cusparseDgtsv.....	257
5.6.11. cusparseCgtsv.....	257
5.6.12. cusparseZgtsv.....	257
5.6.13. cusparseSgtsv2_bufferize.....	258
5.6.14. cusparseDgtsv2_bufferize.....	258
5.6.15. cusparseCgtsv2_bufferize.....	258
5.6.16. cusparseZgtsv2_bufferize.....	258
5.6.17. cusparseSgtsv2.....	258
5.6.18. cusparseDgtsv2.....	259
5.6.19. cusparseCgtsv2.....	259
5.6.20. cusparseZgtsv2.....	259
5.6.21. cusparseSgtsv2_nopivot_bufferize.....	260
5.6.22. cusparseDgtsv2_nopivot_bufferize.....	260
5.6.23. cusparseCgtsv2_nopivot_bufferize.....	260
5.6.24. cusparseZgtsv2_nopivot_bufferize.....	260
5.6.25. cusparseSgtsv2_nopivot.....	260
5.6.26. cusparseDgtsv2_nopivot.....	261
5.6.27. cusparseCgtsv2_nopivot.....	261
5.6.28. cusparseZgtsv2_nopivot.....	261
5.6.29. cusparseSgtsv2StridedBatch_bufferize.....	262
5.6.30. cusparseDgtsv2StridedBatch_bufferize.....	262
5.6.31. cusparseCgtsv2StridedBatch_bufferize.....	262
5.6.32. cusparseZgtsv2StridedBatch_bufferize.....	262
5.6.33. cusparseSgtsv2StridedBatch.....	262
5.6.34. cusparseDgtsv2StridedBatch.....	263
5.6.35. cusparseCgtsv2StridedBatch.....	263
5.6.36. cusparseZgtsv2StridedBatch.....	263
5.6.37. cusparseSgtsvInterleavedBatch_bufferize.....	263
5.6.38. cusparseDgtsvInterleavedBatch_bufferize.....	264
5.6.39. cusparseCgtsvInterleavedBatch_bufferize.....	264
5.6.40. cusparseZgtsvInterleavedBatch_bufferize.....	264

5.6.41. cusparseSgtsvInterleavedBatch.....	264
5.6.42. cusparseDgtsvInterleavedBatch.....	265
5.6.43. cusparseCgtsvInterleavedBatch.....	265
5.6.44. cusparseZgtsvInterleavedBatch.....	265
5.6.45. cusparseSgpsvInterleavedBatch_bufferSize.....	266
5.6.46. cusparseDgpsvInterleavedBatch_bufferSize.....	266
5.6.47. cusparseCgpsvInterleavedBatch_bufferSize.....	266
5.6.48. cusparseZgpsvInterleavedBatch_bufferSize.....	266
5.6.49. cusparseSgpsvInterleavedBatch.....	266
5.6.50. cusparseDgpsvInterleavedBatch.....	267
5.6.51. cusparseCgpsvInterleavedBatch.....	267
5.6.52. cusparseZgpsvInterleavedBatch.....	267
5.6.53. cusparseScsric02_bufferSize.....	268
5.6.54. cusparseDcsric02_bufferSize.....	268
5.6.55. cusparseCcsric02_bufferSize.....	268
5.6.56. cusparseZcsric02_bufferSize.....	268
5.6.57. cusparseScsric02_analysis.....	269
5.6.58. cusparseDcsric02_analysis.....	269
5.6.59. cusparseCcsric02_analysis.....	269
5.6.60. cusparseZcsric02_analysis.....	269
5.6.61. cusparseScsric02.....	270
5.6.62. cusparseDcsric02.....	270
5.6.63. cusparseCcsric02.....	270
5.6.64. cusparseZcsric02.....	270
5.6.65. cusparseXcsric02_zeroPivot.....	271
5.6.66. cusparseScsrilu02_numericBoost.....	271
5.6.67. cusparseDcsrilu02_numericBoost.....	271
5.6.68. cusparseCcsrilu02_numericBoost.....	271
5.6.69. cusparseZcsrilu02_numericBoost.....	271
5.6.70. cusparseScsrilu02_bufferSize.....	272
5.6.71. cusparseDcsrilu02_bufferSize.....	272
5.6.72. cusparseCcsrilu02_bufferSize.....	272
5.6.73. cusparseZcsrilu02_bufferSize.....	272
5.6.74. cusparseScsrilu02_analysis.....	273
5.6.75. cusparseDcsrilu02_analysis.....	273
5.6.76. cusparseCcsrilu02_analysis.....	273
5.6.77. cusparseZcsrilu02_analysis.....	273
5.6.78. cusparseScsrilu02.....	274
5.6.79. cusparseDcsrilu02.....	274
5.6.80. cusparseCcsrilu02.....	274
5.6.81. cusparseZcsrilu02.....	274
5.6.82. cusparseXcsrilu02_zeroPivot.....	275
5.6.83. cusparseSbsric02_bufferSize.....	275

5.6.84. cusparseDbsric02_bufferSize.....	275
5.6.85. cusparseCbsric02_bufferSize.....	275
5.6.86. cusparseZbsric02_bufferSize.....	276
5.6.87. cusparseSbsric02_analysis.....	276
5.6.88. cusparseDbsric02_analysis.....	276
5.6.89. cusparseCbsric02_analysis.....	276
5.6.90. cusparseZbsric02_analysis.....	277
5.6.91. cusparseSbsric02.....	277
5.6.92. cusparseDbsric02.....	277
5.6.93. cusparseCbsric02.....	278
5.6.94. cusparseZbsric02.....	278
5.6.95. cusparseXbsric02_zeroPivot.....	278
5.6.96. cusparseSbsrilu02_numericBoost.....	278
5.6.97. cusparseDbsrilu02_numericBoost.....	279
5.6.98. cusparseCbsrilu02_numericBoost.....	279
5.6.99. cusparseZbsrilu02_numericBoost.....	279
5.6.100. cusparseSbsrilu02_bufferSize.....	279
5.6.101. cusparseDbsrilu02_bufferSize.....	280
5.6.102. cusparseCbsrilu02_bufferSize.....	280
5.6.103. cusparseZbsrilu02_bufferSize.....	280
5.6.104. cusparseSbsrilu02_analysis.....	280
5.6.105. cusparseDbsrilu02_analysis.....	281
5.6.106. cusparseCbsrilu02_analysis.....	281
5.6.107. cusparseZbsrilu02_analysis.....	281
5.6.108. cusparseSbsrilu02.....	281
5.6.109. cusparseDbsrilu02.....	282
5.6.110. cusparseCbsrilu02.....	282
5.6.111. cusparseZbsrilu02.....	282
5.6.112. cusparseXbsrilu02_zeroPivot.....	283
5.7. CUSPARSE Reordering Functions.....	283
5.7.1. cusparseScsrColor.....	283
5.7.2. cusparseDcsrColor.....	283
5.7.3. cusparseCcsrColor.....	284
5.7.4. cusparseZcsrColor.....	284
5.8. CUSPARSE Format Conversion Functions.....	284
5.8.1. cusparseSbsr2csr.....	284
5.8.2. cusparseDbsr2csr.....	285
5.8.3. cusparseCbsr2csr.....	285
5.8.4. cusparseZbsr2csr.....	285
5.8.5. cusparseXcoo2csr.....	285
5.8.6. cusparseScsc2dense.....	286
5.8.7. cusparseDcsc2dense.....	286
5.8.8. cusparseCcsc2dense.....	286



5.8.9. cusparseZcsc2dense.....	286
5.8.10. cusparseScsc2hyb.....	287
5.8.11. cusparseDcsc2hyb.....	287
5.8.12. cusparseCcsc2hyb.....	287
5.8.13. cusparseZcsc2hyb.....	287
5.8.14. cusparseXcsr2bsrNnz.....	288
5.8.15. cusparseScsr2bsr.....	288
5.8.16. cusparseDcsr2bsr.....	288
5.8.17. cusparseCcsr2bsr.....	288
5.8.18. cusparseZcsr2bsr.....	289
5.8.19. cusparseXcsr2coo.....	289
5.8.20. cusparseScsr2csc.....	289
5.8.21. cusparseDcsr2csc.....	289
5.8.22. cusparseCcsr2csc.....	290
5.8.23. cusparseZcsr2csc.....	290
5.8.24. cusparseCsr2cscEx2_bufferSize.....	290
5.8.25. cusparseCsr2cscEx2.....	290
5.8.26. cusparseScsr2dense.....	291
5.8.27. cusparseDcsr2dense.....	291
5.8.28. cusparseCcsr2dense.....	291
5.8.29. cusparseZcsr2dense.....	291
5.8.30. cusparseScsr2hyb.....	292
5.8.31. cusparseDcsr2hyb.....	292
5.8.32. cusparseCcsr2hyb.....	292
5.8.33. cusparseZcsr2hyb.....	292
5.8.34. cusparseSdense2csc.....	293
5.8.35. cusparseDdense2csc.....	293
5.8.36. cusparseCdense2csc.....	293
5.8.37. cusparseZdense2csc.....	293
5.8.38. cusparseSdense2csr.....	293
5.8.39. cusparseDdense2csr.....	294
5.8.40. cusparseCdense2csr.....	294
5.8.41. cusparseZdense2csr.....	294
5.8.42. cusparseSdense2hyb.....	294
5.8.43. cusparseDdense2hyb.....	294
5.8.44. cusparseCdense2hyb.....	295
5.8.45. cusparseZdense2hyb.....	295
5.8.46. cusparseShyb2csc.....	295
5.8.47. cusparseDhyb2csc.....	295
5.8.48. cusparseChyb2csc.....	295
5.8.49. cusparseZhyb2csc.....	296
5.8.50. cusparseShyb2csr.....	296
5.8.51. cusparseDhyb2csr.....	296

5.8.52. cusparseChyb2csr.....	296
5.8.53. cusparseZhyb2csr.....	297
5.8.54. cusparseShyb2dense.....	297
5.8.55. cusparseDhyb2dense.....	297
5.8.56. cusparseChyb2dense.....	297
5.8.57. cusparseZhyb2dense.....	297
5.8.58. cusparseSnnz.....	298
5.8.59. cusparseDnnz.....	298
5.8.60. cusparseCnnz.....	298
5.8.61. cusparseZnnz.....	298
5.8.62. cusparseSgebsr2gebsc_bufferSize.....	299
5.8.63. cusparseDgebsr2gebsc_bufferSize.....	299
5.8.64. cusparseCgebsr2gebsc_bufferSize.....	299
5.8.65. cusparseZgebsr2gebsc_bufferSize.....	299
5.8.66. cusparseSgebsr2gebsc.....	299
5.8.67. cusparseDgebsr2gebsc.....	300
5.8.68. cusparseCgebsr2gebsc.....	300
5.8.69. cusparseZgebsr2gebsc.....	300
5.8.70. cusparseSgebsr2gebsr_bufferSize.....	301
5.8.71. cusparseDgebsr2gebsr_bufferSize.....	301
5.8.72. cusparseCgebsr2gebsr_bufferSize.....	301
5.8.73. cusparseZgebsr2gebsr_bufferSize.....	301
5.8.74. cusparseXgebsr2gebsrNnz.....	302
5.8.75. cusparseSgebsr2gebsr.....	302
5.8.76. cusparseDgebsr2gebsr.....	302
5.8.77. cusparseCgebsr2gebsr.....	303
5.8.78. cusparseZgebsr2gebsr.....	303
5.8.79. cusparseSgebsr2csr.....	303
5.8.80. cusparseDgebsr2csr.....	304
5.8.81. cusparseCgebsr2csr.....	304
5.8.82. cusparseZgebsr2csr.....	304
5.8.83. cusparseScsr2gebsr_bufferSize.....	304
5.8.84. cusparseDcsr2gebsr_bufferSize.....	305
5.8.85. cusparseCcsr2gebsr_bufferSize.....	305
5.8.86. cusparseZcsr2gebsr_bufferSize.....	305
5.8.87. cusparseXcsr2gebsrNnz.....	305
5.8.88. cusparseScsr2gebsr.....	306
5.8.89. cusparseDcsr2gebsr.....	306
5.8.90. cusparseCcsr2gebsr.....	306
5.8.91. cusparseZcsr2gebsr.....	307
5.8.92. cusparseCreatelIdentityPermutation.....	307
5.8.93. cusparseXcoosort_bufferSize.....	307
5.8.94. cusparseXcoosortByRow.....	307

5.8.95. cusparseXcoosortByColumn.....	307
5.8.96. cusparseXcsrsort_bufferSize.....	308
5.8.97. cusparseXcsrsort.....	308
5.8.98. cusparseXcscsort_bufferSize.....	308
5.8.99. cusparseXcscsort.....	308
5.8.100. cusparseScsru2csr_bufferSize.....	308
5.8.101. cusparseDcsru2csr_bufferSize.....	309
5.8.102. cusparseCcsru2csr_bufferSize.....	309
5.8.103. cusparseZcsru2csr_bufferSize.....	309
5.8.104. cusparseScsru2csr.....	309
5.8.105. cusparseDcsru2csr.....	309
5.8.106. cusparseCcsru2csr.....	310
5.8.107. cusparseZcsru2csr.....	310
5.8.108. cusparseScsr2csru.....	310
5.8.109. cusparseDcsr2csru.....	310
5.8.110. cusparseCcsr2csru.....	311
5.8.111. cusparseZcsr2csru.....	311
5.9. CUSPARSE Generic API Functions.....	311
5.9.1. cusparseDenseToSparse_bufferSize.....	311
5.9.2. cusparseDenseToSparse_analysis.....	311
5.9.3. cusparseDenseToSparse_convert.....	312
5.9.4. cusparseSparseToDense_bufferSize.....	312
5.9.5. cusparseSparseToDense.....	312
5.9.6. cusparseCreateSpVec.....	312
5.9.7. cusparseDestroySpVec.....	312
5.9.8. cusparseSpVecGet.....	313
5.9.9. cusparseSpVecGetIndexBase.....	313
5.9.10. cusparseSpVecGetValues.....	313
5.9.11. cusparseSpVecSetValues.....	313
5.9.12. cusparseCreateDnVec.....	313
5.9.13. cusparseDestroyDnVec.....	314
5.9.14. cusparseDnVecGet.....	314
5.9.15. cusparseDnVecGetValues.....	314
5.9.16. cusparseDnVecSetValues.....	314
5.9.17. cusparseCreateCoo.....	314
5.9.18. cusparseCreateCooAoS.....	315
5.9.19. cusparseCreateCsr.....	315
5.9.20. cusparseCreateBlockedEll.....	315
5.9.21. cusparseDestroySpMat.....	315
5.9.22. cusparseCooGet.....	316
5.9.23. cusparseCooAoSGet.....	316
5.9.24. cusparseCsrGet.....	316
5.9.25. cusparseBlockedEllGet.....	316

5.9.26.	cusparseCsrSetPointers.....	316
5.9.27.	cusparseCscSetPointers.....	317
5.9.28.	cusparseSpMatGetFormat.....	317
5.9.29.	cusparseSpMatGetIndexBase.....	317
5.9.30.	cusparseSpMatGetSize.....	317
5.9.31.	cusparseSpMatGetValues.....	317
5.9.32.	cusparseSpMatSetValues.....	317
5.9.33.	cusparseSpMatGetStridedBatch.....	318
5.9.34.	cusparseSpMatSetStridedBatch.....	318
5.9.35.	cusparseSpMatGetAttribute.....	318
5.9.36.	cusparseSpMatSetAttribute.....	318
5.9.37.	cusparseCreateDnMat.....	318
5.9.38.	cusparseDestroyDnMat.....	319
5.9.39.	cusparseDnMatGet.....	319
5.9.40.	cusparseDnMatGetValues.....	319
5.9.41.	cusparseDnMatSetValues.....	319
5.9.42.	cusparseDnMatGetStridedBatch.....	319
5.9.43.	cusparseDnMatSetStridedBatch.....	319
5.9.44.	cusparseSpVV_bufferSize.....	320
5.9.45.	cusparseSpVV.....	320
5.9.46.	cusparseSpMV_bufferSize.....	320
5.9.47.	cusparseSpMV.....	320
5.9.48.	cusparseSpSV_CreateDescr.....	321
5.9.49.	cusparseSpSV_DestroyDescr.....	321
5.9.50.	cusparseSpSV_bufferSize.....	321
5.9.51.	cusparseSpSV_analysis.....	321
5.9.52.	cusparseSpSV_solve.....	321
5.9.53.	cusparseSpMM_bufferSize.....	322
5.9.54.	cusparseSpMM_preprocess.....	322
5.9.55.	cusparseSpMM.....	322
5.9.56.	cusparseSpSM_CreateDescr.....	323
5.9.57.	cusparseSpSM_DestroyDescr.....	323
5.9.58.	cusparseSpSM_bufferSize.....	323
5.9.59.	cusparseSpSM_analysis.....	323
5.9.60.	cusparseSpSM_solve.....	323
5.9.61.	cusparseSDDMM_bufferSize.....	324
5.9.62.	cusparseSDDMM_preprocess.....	324
5.9.63.	cusparseSDDMM.....	324
5.9.64.	cusparseSpGEMM_CreateDescr.....	324
5.9.65.	cusparseSpGEMM_DestroyDescr.....	325
5.9.66.	cusparseSpGEMM_workEstimation.....	325
5.9.67.	cusparseSpGEMM_compute.....	325
5.9.68.	cusparseSpGEMM_copy.....	325

5.9.69. cusparseSpGEMMreuse_workEstimation.....	326
5.9.70. cusparseSpGEMMreuse_nnz.....	327
5.9.71. cusparseSpGEMMreuse_copy.....	327
5.9.72. cusparseSpGEMMreuse_compute.....	328
Chapter 6. Matrix Solver Runtime Library APIs.....	329
6.1. CUSOLVER Definitions and Helper Functions.....	329
6.1.1. cusolverDnCreate.....	331
6.1.2. cusolverDnDestroy.....	331
6.1.3. cusolverDnCreateParams.....	331
6.1.4. cusolverDnDestroyParams.....	331
6.1.5. cusolverDnSetAdvOptions.....	332
6.1.6. cusolverDnGetStream.....	332
6.1.7. cusolverDnSetStream.....	332
6.2. cusolverDn Legacy API.....	332
6.2.1. cusolverDnSpotrf_buffersize.....	332
6.2.2. cusolverDnDpotrf_buffersize.....	332
6.2.3. cusolverDnCpotrf_buffersize.....	333
6.2.4. cusolverDnZpotrf_buffersize.....	333
6.2.5. cusolverDnSpotrf.....	333
6.2.6. cusolverDnDpotrf.....	333
6.2.7. cusolverDnCpotrf.....	333
6.2.8. cusolverDnZpotrf.....	334
6.2.9. cusolverDnSpotrs.....	334
6.2.10. cusolverDnDpotrs.....	334
6.2.11. cusolverDnCpotrs.....	334
6.2.12. cusolverDnZpotrs.....	335
6.2.13. cusolverDnSpotrfBatched.....	335
6.2.14. cusolverDnDpotrfBatched.....	335
6.2.15. cusolverDnCpotrfBatched.....	335
6.2.16. cusolverDnZpotrfBatched.....	335
6.2.17. cusolverDnSpotrsBatched.....	336
6.2.18. cusolverDnDpotrsBatched.....	336
6.2.19. cusolverDnCpotrsBatched.....	336
6.2.20. cusolverDnZpotrsBatched.....	336
6.2.21. cusolverDnSpotri_buffersize.....	337
6.2.22. cusolverDnDpotri_buffersize.....	337
6.2.23. cusolverDnCpotri_buffersize.....	337
6.2.24. cusolverDnZpotri_buffersize.....	337
6.2.25. cusolverDnSpotri.....	338
6.2.26. cusolverDnDpotri.....	338
6.2.27. cusolverDnCpotri.....	338
6.2.28. cusolverDnZpotri.....	338
6.2.29. cusolverDnStrtri_buffersize.....	339

6.2.30. cusolverDnDtrtri_buffersize.....	339
6.2.31. cusolverDnCtrtri_buffersize.....	339
6.2.32. cusolverDnZtrtri_buffersize.....	339
6.2.33. cusolverDnStrtri.....	339
6.2.34. cusolverDnDtrtri.....	340
6.2.35. cusolverDnCtrtri.....	340
6.2.36. cusolverDnZtrtri.....	340
6.2.37. cusolverDnSlauum_buffersize.....	340
6.2.38. cusolverDnDlauum_buffersize.....	341
6.2.39. cusolverDnClauum_buffersize.....	341
6.2.40. cusolverDnZlauum_buffersize.....	341
6.2.41. cusolverDnSlauum.....	341
6.2.42. cusolverDnDlauum.....	341
6.2.43. cusolverDnClauum.....	342
6.2.44. cusolverDnZlauum.....	342
6.2.45. cusolverDnSgetrf_buffersize.....	342
6.2.46. cusolverDnDgetrf_buffersize.....	342
6.2.47. cusolverDnCgetrf_buffersize.....	343
6.2.48. cusolverDnZgetrf_buffersize.....	343
6.2.49. cusolverDnSgetrf.....	343
6.2.50. cusolverDnDgetrf.....	343
6.2.51. cusolverDnCgetrf.....	343
6.2.52. cusolverDnZgetrf.....	344
6.2.53. cusolverDnSgetrs.....	344
6.2.54. cusolverDnDgetrs.....	344
6.2.55. cusolverDnCgetrs.....	344
6.2.56. cusolverDnZgetrs.....	345
6.2.57. cusolverDnSlaswp.....	345
6.2.58. cusolverDnDlaswp.....	345
6.2.59. cusolverDnClaswp.....	345
6.2.60. cusolverDnZlaswp.....	345
6.2.61. cusolverDnSgeqrf_buffersize.....	346
6.2.62. cusolverDnDgeqrf_buffersize.....	346
6.2.63. cusolverDnCgeqrf_buffersize.....	346
6.2.64. cusolverDnZgeqrf_buffersize.....	346
6.2.65. cusolverDnSgeqrf.....	346
6.2.66. cusolverDnDgeqrf.....	347
6.2.67. cusolverDnCgeqrf.....	347
6.2.68. cusolverDnZgeqrf.....	347
6.2.69. cusolverDnSorgqr_buffersize.....	347
6.2.70. cusolverDnDorgqr_buffersize.....	347
6.2.71. cusolverDnCorgqr_buffersize.....	348
6.2.72. cusolverDnZorgqr_buffersize.....	348

6.2.73. cusolverDnSorgqr.....	348
6.2.74. cusolverDnDorgqr.....	348
6.2.75. cusolverDnCorgqr.....	348
6.2.76. cusolverDnZorgqr.....	349
6.2.77. cusolverDnSormqr_buffersize.....	349
6.2.78. cusolverDnDormqr_buffersize.....	349
6.2.79. cusolverDnCormqr_buffersize.....	349
6.2.80. cusolverDnZormqr_buffersize.....	350
6.2.81. cusolverDnSormqr.....	350
6.2.82. cusolverDnDormqr.....	350
6.2.83. cusolverDnCormqr.....	350
6.2.84. cusolverDnZormqr.....	351
6.2.85. cusolverDnSsytrf_buffersize.....	351
6.2.86. cusolverDnDsytrf_buffersize.....	351
6.2.87. cusolverDnCsytrf_buffersize.....	351
6.2.88. cusolverDnZsytrf_buffersize.....	351
6.2.89. cusolverDnSsytrf.....	352
6.2.90. cusolverDnDsytrf.....	352
6.2.91. cusolverDnCsytrf.....	352
6.2.92. cusolverDnZsytrf.....	352
6.2.93. cusolverDnSsytrs_bufferSize.....	353
6.2.94. cusolverDnDsytrs_bufferSize.....	353
6.2.95. cusolverDnCsytrs_bufferSize.....	353
6.2.96. cusolverDnZsytrs_bufferSize.....	353
6.2.97. cusolverDnSsytrs.....	354
6.2.98. cusolverDnDsytrs.....	354
6.2.99. cusolverDnCsytrs.....	354
6.2.100. cusolverDnZsytrs.....	354
6.2.101. cusolverDnSsytri_bufferSize.....	355
6.2.102. cusolverDnDsytri_bufferSize.....	355
6.2.103. cusolverDnCsytri_bufferSize.....	355
6.2.104. cusolverDnZsytri_bufferSize.....	355
6.2.105. cusolverDnSsytri.....	356
6.2.106. cusolverDnDsytri.....	356
6.2.107. cusolverDnCsytri.....	356
6.2.108. cusolverDnZsytri.....	356
6.3. cusolverDn Legacy Eigenvalue Solver API.....	357
6.3.1. cusolverDnSgebrd_buffersize.....	357
6.3.2. cusolverDnDgebrd_buffersize.....	357
6.3.3. cusolverDnCgebrd_buffersize.....	357
6.3.4. cusolverDnZgebrd_buffersize.....	357
6.3.5. cusolverDnSgebrd.....	357
6.3.6. cusolverDnDgebrd.....	358

6.3.7. cusolverDnCgebrd.....	358
6.3.8. cusolverDnZgebrd.....	358
6.3.9. cusolverDnSorgbr_buffersize.....	359
6.3.10. cusolverDnDorgbr_buffersize.....	359
6.3.11. cusolverDnCungbr_buffersize.....	359
6.3.12. cusolverDnZungbr_buffersize.....	359
6.3.13. cusolverDnSorgbr.....	360
6.3.14. cusolverDnDorgbr.....	360
6.3.15. cusolverDnCungbr.....	360
6.3.16. cusolverDnZungbr.....	360
6.3.17. cusolverDnSsytrd_buffersize.....	361
6.3.18. cusolverDnDsytrd_buffersize.....	361
6.3.19. cusolverDnChetrd_buffersize.....	361
6.3.20. cusolverDnZhetrd_buffersize.....	361
6.3.21. cusolverDnSsytrd.....	362
6.3.22. cusolverDnDsytrd.....	362
6.3.23. cusolverDnChetrd.....	362
6.3.24. cusolverDnZhetrd.....	362
6.3.25. cusolverDnSormtr_buffersize.....	363
6.3.26. cusolverDnDormtr_buffersize.....	363
6.3.27. cusolverDnCunmtr_buffersize.....	363
6.3.28. cusolverDnZumtr_buffersize.....	363
6.3.29. cusolverDnSormtr.....	364
6.3.30. cusolverDnDormtr.....	364
6.3.31. cusolverDnCunmtr.....	364
6.3.32. cusolverDnZumtr.....	364
6.3.33. cusolverDnSorgtr_buffersize.....	365
6.3.34. cusolverDnDorgtr_buffersize.....	365
6.3.35. cusolverDnCungtr_buffersize.....	365
6.3.36. cusolverDnZungtr_buffersize.....	365
6.3.37. cusolverDnSorgtr.....	366
6.3.38. cusolverDnDorgtr.....	366
6.3.39. cusolverDnCungtr.....	366
6.3.40. cusolverDnZungtr.....	366
6.3.41. cusolverDnSgesvd_buffersize.....	367
6.3.42. cusolverDnDgesvd_buffersize.....	367
6.3.43. cusolverDnCgesvd_buffersize.....	367
6.3.44. cusolverDnZgesvd_buffersize.....	367
6.3.45. cusolverDnSgesvd.....	367
6.3.46. cusolverDnDgesvd.....	368
6.3.47. cusolverDnCgesvd.....	368
6.3.48. cusolverDnZgesvd.....	368
6.3.49. cusolverDnSsyevd_buffersize.....	369



6.3.50. cusolverDnDsyevd_buffersize.....	369
6.3.51. cusolverDnCheevd_buffersize.....	369
6.3.52. cusolverDnZheevd_buffersize.....	369
6.3.53. cusolverDnSsyevd.....	370
6.3.54. cusolverDnDsyevd.....	370
6.3.55. cusolverDnCheevd.....	370
6.3.56. cusolverDnZheevd.....	370
6.3.57. cusolverDnSsyevdx_buffersize.....	371
6.3.58. cusolverDnDsyevdx_buffersize.....	371
6.3.59. cusolverDnCheevdx_buffersize.....	371
6.3.60. cusolverDnZheevdx_buffersize.....	371
6.3.61. cusolverDnSsyevdx.....	372
6.3.62. cusolverDnDsyevdx.....	372
6.3.63. cusolverDnCheevdx.....	372
6.3.64. cusolverDnZheevdx.....	372
6.3.65. cusolverDnSsygvd_buffersize.....	373
6.3.66. cusolverDnDsygvd_buffersize.....	373
6.3.67. cusolverDnChegvd_buffersize.....	373
6.3.68. cusolverDnZhegvd_buffersize.....	374
6.3.69. cusolverDnSsygvd.....	374
6.3.70. cusolverDnDsygvd.....	374
6.3.71. cusolverDnChegvd.....	374
6.3.72. cusolverDnZhegvd.....	375
6.3.73. cusolverDnSsygvdx_buffersize.....	375
6.3.74. cusolverDnDsygvdx_buffersize.....	375
6.3.75. cusolverDnChegvdx_buffersize.....	375
6.3.76. cusolverDnZhegvdx_buffersize.....	376
6.3.77. cusolverDnSsygvdx.....	376
6.3.78. cusolverDnDsygvdx.....	376
6.3.79. cusolverDnChegvdx.....	377
6.3.80. cusolverDnZhegvdx.....	377
6.4. cusolverDn 64-bit API.....	377
6.4.1. cusolverDnXpotrf_buffersize.....	377
6.4.2. cusolverDnXpotrf.....	378
6.4.3. cusolverDnXpotrs.....	378
6.4.4. cusolverDnXgeqrf_buffersize.....	378
6.4.5. cusolverDnXgeqrf.....	379
6.4.6. cusolverDnXgetrf_buffersize.....	379
6.4.7. cusolverDnXgetrf.....	379
6.4.8. cusolverDnXgetrs.....	380
6.4.9. cusolverDnXsyevd_buffersize.....	380
6.4.10. cusolverDnXsyevd.....	381
6.4.11. cusolverDnXsyevdx_buffersize.....	381

6.4.12. cusolverDnXsyevdX.....	382
6.4.13. cusolverDnXgesvd_buffersize.....	382
6.4.14. cusolverDnXgesvd.....	383
6.4.15. cusolverDnXgesvdp_buffersize.....	383
6.4.16. cusolverDnXgesvdp.....	384
6.5. NVLAMATH Runtime Library.....	384
6.5.1. NVLAMATH Automatic Drop-In Acceleration.....	385
6.5.2. NVLAMATH Usage From CUDA Fortran.....	386
6.5.3. NVLAMATH Usage From OpenACC.....	386
6.5.4. NVLAMATH Usage From OpenMP.....	387
6.5.5. NVLAMATH List of Current Subroutines.....	388
6.5.6. NVLAMATH Argument Checks and CPU Fallback.....	388
Chapter 7. Tensor Primitives Runtime Library APIs.....	390
7.1. CUTENSOR Definitions and Helper Functions.....	390
7.1.1. cutensorInit.....	393
7.1.2. cutensorInitTensorDescriptor.....	393
7.1.3. cutensorGetAlignmentRequirement.....	393
7.1.4. cutensorGetErrorString.....	393
7.1.5. cutensorGetVersion.....	393
7.1.6. cutensorGetCudartVersion.....	394
7.2. CUTENSOR Element-wise Operations.....	394
7.2.1. cutensorPermutation.....	394
7.2.2. cutensorElementwiseBinary.....	394
7.2.3. cutensorElementwiseTrinary.....	395
7.3. CUTENSOR Reduction Operations.....	395
7.3.1. cutensorReductionGetWorkspace.....	395
7.3.2. cutensorReduction.....	395
7.4. CUTENSOR Contraction Operations.....	396
7.4.1. cutensorInitContractionDescriptor.....	396
7.4.2. cutensorInitContractionFind.....	396
7.4.3. cutensorInitContractionPlan.....	396
7.4.4. cutensorContractionGetWorkspace.....	397
7.4.5. cutensorContractionMaxAlgos.....	397
7.4.6. cutensorContraction.....	397
7.5. CUTENSOR Fortran Extensions.....	397
7.5.1. Fortran Reshape.....	398
7.5.2. Fortran Transpose.....	398
7.5.3. Fortran Spread.....	398
7.5.4. Fortran Element-wise Expressions.....	399
7.5.5. Fortran Matmul Operations.....	400
7.5.6. Supported Element-wise Functions.....	400
Chapter 8. NVIDIA Collective Communications Library (NCCL) APIs.....	402
8.1. NCCL Definitions and Helper Functions.....	402

8.1.1. ncclGetVersion.....	403
8.1.2. ncclGetUniqueld.....	403
8.1.3. ncclCommInitRank.....	404
8.1.4. ncclCommInitAll.....	404
8.1.5. ncclCommDestroy.....	404
8.1.6. ncclCommAbort.....	404
8.1.7. ncclGetErrorString.....	404
8.1.8. ncclCommGetAsyncError.....	404
8.1.9. ncclCommCount.....	405
8.1.10. ncclCommCuDevice.....	405
8.1.11. ncclCommUserRank.....	405
8.2. NCCL Collective Communication Functions.....	405
8.2.1. ncclAllReduce.....	405
8.2.2. ncclBroadcast.....	406
8.2.3. ncclReduce.....	406
8.2.4. ncclAllGather.....	406
8.2.5. ncclReduceScatter.....	407
8.3. NCCL Point To Point Communication Functions.....	407
8.3.1. ncclSend.....	407
8.3.2. ncclRecv.....	407
8.4. NCCL Group Calls.....	408
8.4.1. ncclGroupStart.....	408
8.4.2. ncclGroupEnd.....	408
Chapter 9. NVSHMEM Communication Library APIs.....	409
9.1. NVSHMEM Definitions, Setup, Exit, and Query Functions.....	409
9.1.1. nvshmem_init.....	411
9.1.2. nvshmemx_init_attr.....	411
9.1.3. nvshmem_my_pe.....	411
9.1.4. nvshmem_n_pes.....	411
9.1.5. nvshmem_team_my_pe.....	411
9.1.6. nvshmem_team_n_pes.....	411
9.1.7. nvshmem_team_get_config.....	412
9.1.8. nvshmem_team_translate_pe.....	412
9.1.9. nvshmem_team_split_strided.....	412
9.1.10. nvshmem_team_split_2d.....	412
9.1.11. nvshmem_team_destroy.....	412
9.1.12. nvshmem_info_get_version.....	412
9.1.13. nvshmem_info_get_name.....	413
9.1.14. nvshmem_finalize.....	413
9.1.15. nvshmem_ptr.....	413
9.2. NVSHMEM Memory Management Functions.....	413
9.2.1. nvshmem_malloc.....	414
9.2.2. nvshmem_free.....	414

9.2.3. nvshmem_align.....	414
9.2.4. nvshmem_calloc.....	414
9.3. NVSHMEM Remote Memory Access Functions.....	414
9.3.1. nvshmem_put.....	415
9.3.2. nvshmem_p.....	417
9.3.3. nvshmem_iput.....	418
9.3.4. nvshmem_put_nbi.....	421
9.3.5. nvshmemx_put_block.....	423
9.3.6. nvshmemx_put_warp.....	425
9.3.7. nvshmem_get.....	426
9.3.8. nvshmem_g.....	428
9.3.9. nvshmem_iget.....	429
9.3.10. nvshmem_get_nbi.....	432
9.3.11. nvshmemx_get_block.....	434
9.3.12. nvshmemx_get_warp.....	436
9.4. NVSHMEM Collective Communication Functions.....	437
9.4.1. nvshmem_barrier, nvshmem_barrier_all.....	437
9.4.2. nvshmem_sync, nvshmem_sync_all.....	438
9.4.3. nvshmem_alltoall.....	438
9.4.4. nvshmem_broadcast.....	440
9.4.5. nvshmem_collect.....	443
9.4.6. NVSHMEM Reductions.....	445
9.4.6.1. nvshmem_and_reduce.....	445
9.4.6.2. nvshmem_or_reduce.....	447
9.4.6.3. nvshmem_xor_reduce.....	448
9.4.6.4. nvshmem_max_reduce.....	450
9.4.6.5. nvshmem_min_reduce.....	452
9.4.6.6. nvshmem_sum_reduce.....	454
9.4.6.7. nvshmem_prod_reduce.....	456
9.5. NVSHMEM Point to Point Synchronization Functions.....	458
9.5.1. nvshmem_wait_until.....	458
9.6. NVSHMEM Memory Ordering Functions.....	459
9.6.1. nvshmem_fence.....	459
9.6.2. nvshmem_quiet.....	459
Chapter 10. NVTX Profiling Library APIs.....	460
10.1. NVTX Basic Tooling APIs.....	460
10.1.1. nvtxStartRange.....	461
10.1.2. nvtxEndRange.....	461
10.2. NVTX Advanced Tooling APIs.....	461
10.2.1. NVTX Definitions and Derived Types.....	461
10.2.2. nvtxInitialize.....	462
10.2.3. nvtxDomainCreate.....	463
10.2.4. nvtxDomainDestroy.....	463

10.2.5. nvtxDomainRegisterString.....	463
10.2.6. nvtxDomainNameCategory.....	463
10.2.7. nvtxNameCategory.....	463
10.2.8. nvtxDomainMarkEx.....	464
10.2.9. nvtxMarkEx.....	464
10.2.10. nvtxMark.....	464
10.2.11. nvtxDomainRangeStartEx.....	464
10.2.12. nvtxRangeStartEx.....	464
10.2.13. nvtxRangeStart.....	464
10.2.14. nvtxDomainRangeEnd.....	465
10.2.15. nvtxRangeEnd.....	465
10.2.16. nvtxDomainRangePushEx.....	465
10.2.17. nvtxRangePushEx.....	465
10.2.18. nvtxRangePush.....	465
10.2.19. nvtxDomainRangePop.....	465
10.2.20. nvtxRangePop.....	466
10.3. NVTX Automated Instrumentation.....	466
Chapter 11. Examples.....	467
11.1. Using cuBLAS from OpenACC Host Code.....	467
11.2. Using cuBLAS from CUDA Fortran Host Code.....	469
11.3. Using cuFFT from OpenACC Host Code.....	471
11.4. Using cuFFT from CUDA Fortran Host Code.....	472
11.5. Using cufftXt from CUDA Fortran Host Code.....	473
11.6. Using cuFFTMp from either OpenACC or CUDA Fortran.....	478
11.7. Using cuRAND from OpenACC Host Code.....	485
11.8. Using cuRAND from OpenACC Device Code.....	487
11.9. Using cuRAND from CUDA Fortran Host Code.....	488
11.10. Using cuRAND from CUDA Fortran Device Code.....	490
11.11. Using cuSPARSE from OpenACC Host Code.....	491
11.12. Using cuSPARSE from CUDA Fortran Host Code.....	493
11.13. Using cuTENSOR from CUDA Fortran Host Code.....	495
11.14. Using cuTENSOREX from CUDA Fortran Host Code.....	496
11.15. Using cuTENSOR from OpenACC Host Code.....	497

## LIST OF TABLES

Table 1 NVLAMATH List of Subroutines .....	388
--	-----

# PREFACE

This document describes the NVIDIA Fortran interfaces to cuBLAS, cuFFT, cuRAND, and cuSPARSE, which are 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 APIs**

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

### 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.

### **FORTTRAN**

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 test 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](https://docs.nvidia.com/hpc-sdk/definitions) at docs.nvidia.com/hpc-sdk/definitions.

## 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 Guide](#)



- ▶ [HPC Compilers User Guide, docs.nvidia.com/hpc-sdk/compilers/pdf/hpc227ug.pdf](https://docs.nvidia.com/hpc-sdk/compilers/pdf/hpc227ug.pdf)



# Chapter 1.

## INTRODUCTION

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 library.
- ▶ NVSHMEM, a library implementation of OpenSHMEM on GPUs.

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, <http://www.openacc.org>](http://www.openacc.org) 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`.

## 1.1. 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 issues 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.

## 1.2. 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
!$acc kernels
c = c / (m*n)
!$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.

### 1.3. 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.

## 1.4. 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](#).

## 1.5. 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

```

## 1.6. 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 runtime 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.

## 1.7. 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 transformations 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
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

```

## 1.8. 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]`: 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 2.

## BLAS RUNTIME APIS

This section describes the Fortran interfaces to the CUDA BLAS libraries. There are currently three 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.

NVIDIA currently ships with three 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.

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)`.

## 2.1. 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

```

### 2.1.1. 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` occurrences. 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

```

### 2.1.2. 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` occurrences.

```

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

```

### 2.1.3. cublasGetVersion

This function returns the version number of the cuBLAS library.

```

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

```

### 2.1.4. 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
```

### 2.1.5. 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
```

### 2.1.6. cublasGetStatusName

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

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

### 2.1.7. cublasGetStatusString

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

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

### 2.1.8. 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
```

### 2.1.9. 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
```

## 2.1.10. cublasGetAtomicsMode

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

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

## 2.1.11. 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
```

## 2.1.12. cublasGetMathMode

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

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

## 2.1.13. 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
```

## 2.1.14. 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
```

## 2.1.15. 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 cublasvector(n, elemsize, x, incx, y, incy)
  integer :: n, elemsize, incx, incy
  integer*1, dimension(*) :: x
  integer*1, device, dimension(*) :: y
```

### 2.1.16. 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
```

### 2.1.17. 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
```

### 2.1.18. 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
```

### 2.1.19. 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
```

## 2.1.20. 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
```

## 2.1.21. 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
```

## 2.1.22. 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
```

## 2.2. Single Precision Functions and Subroutines

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

### 2.2.1. 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
```

### 2.2.2. 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
```

### 2.2.3. 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

```

## 2.2.4. 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

```

## 2.2.5. 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

```

## 2.2.6. 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

```

## 2.2.7. 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

```

## 2.2.8. 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

```

## 2.2.9. 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

```

## 2.2.10. srotm

SROTMM 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 SROTMMG 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
```

## 2.2.11. srotmg

SROTMMG 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.E0, -1.E0, or 0.E0 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
```

## 2.2.12. 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
```

## 2.2.13. 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
```

## 2.2.14. 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
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
```

## 2.2.15. 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
```

## 2.2.16. 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
```

## 2.2.17. 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

```

## 2.2.18. 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

```

## 2.2.19. 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
```

## 2.2.20. 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
  integer :: t
  integer :: n, incx, incy
  real(4), device, dimension(*) :: a, x, y
  real(4), device :: alpha ! device or host variable
```

## 2.2.21. 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
```

## 2.2.22. 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

```

## 2.2.23. 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

```

## 2.2.24. 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

```

## 2.2.25. 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

```

## 2.2.26. 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
  integer :: u, t, d
  integer :: n, incx
  real(4), device, dimension(*) :: a, x

```

## 2.2.27. 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
```

## 2.2.28. 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
```

## 2.2.29. 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
```

## 2.2.30. sgemm

SGEMM performs one of the matrix-matrix operations  $C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
  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
```

## 2.2.31. ssymm

SSYMM performs one of the matrix-matrix operations  $C := \alpha \cdot A \cdot B + \beta \cdot C$ , or  $C := \alpha \cdot B \cdot A + \beta \cdot C$ , 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
```

### 2.2.32. 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
```

### 2.2.33. 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
```

## 2.2.34. ssyrkx

SSYRKX performs a variation of the symmetric rank k update  $C := \alpha * A * B * B^T + \beta * C$ , 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
```

## 2.2.35. strmm

STRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^T$ .

```
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
```



## 2.2.36. strsm

STRSM solves one of the matrix equations  $\text{op}(A)X = \alpha B$ , or  $X\text{op}(A) = \alpha B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^T$ . 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
```

## 2.2.37. cublasSgemvBatched

SGEMV performs a batch of the matrix-vector operations  $Y := \alpha \text{op}(A) * X + \beta Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^T$ ,  $\alpha$  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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devpstr), 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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: xarray(*)
  integer :: incx
  real(4), device :: beta ! device or host variable
  type(c_devpstr), device :: yarray(*)
  integer :: incy
```

```
integer :: batchSize
```

## 2.2.38. cublasSgemvBatched

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

```
integer(4) function cublasSgemvBatched(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchSize)
  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
  type(c_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devpstr), device :: Carray(*)
  integer :: ldc
  integer :: batchSize
```

```
integer(4) function cublasSgemvBatched_v2(h, transa, transb, m, n, k, alpha,
Aarray, lda, Barray, ldb, beta, Carray, ldc, batchSize)
  type(cublasHandle) :: h
  integer :: transa
  integer :: transb
  integer :: m, n, k
  real(4), device :: alpha ! device or host variable
  type(c_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Barray(*)
  integer :: ldb
  real(4), device :: beta ! device or host variable
  type(c_devpstr), device :: Carray(*)
  integer :: ldc
  integer :: batchSize
```

## 2.2.39. 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  $\text{TRANS} = 'N'$  and  $m \geq n$ : find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize  $\|B - A * X\|$ . 2. If  $\text{TRANS} = 'N'$  and  $m < n$ : find the minimum norm solution of an underdetermined system  $A * X = B$ . 3. If  $\text{TRANS} = 'T'$  and  $m \geq n$ : find the minimum norm solution of an undetermined system  $A^{**T} * X = B$ . 4. If  $\text{TRANS} = 'T'$  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- $\text{NRHS}$  right hand side matrix  $B$  and the  $N$ -by- $\text{NRHS}$  solution matrix  $X$ .

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

```

integer :: lda
type(c_devpstr), device :: Carray(*)
integer :: ldc
integer :: info(*)
integer, device :: devinfo(*)
integer :: batchSize

```

## 2.2.40. 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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Tau(*)
  integer :: info(*)
  integer :: batchSize

```

## 2.2.41. 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_devpstr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchSize

```

## 2.2.42. cublasSgetriBatched

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

```

integer(4) function cublasSgetriBatched(h, n, Aarray, lda, ipvt, Carray, ldc,
info, batchSize)
  type(cublasHandle) :: h
  integer :: n
  type(c_devpstr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devpstr), device :: Carray(*)
  integer :: ldc
  integer, device :: info(*)
  integer :: batchSize

```

## 2.2.43. 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devp_ptr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

## 2.2.44. 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  type(c_devp_ptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

## 2.2.45. 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
  type(c_devp_ptr), device :: A(*)
  integer :: lda
  type(c_devp_ptr), 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 :: batchSize

```

## 2.2.46. cublasSgemvStridedBatched

SGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  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, batchSize)
  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 :: batchSize

```

```

integer(4) function cublasSgemvStridedBatched_v2(h, trans, m, n, alpha, &
      A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchSize)
  type(cublasHandle) :: h
  integer :: trans
  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 :: batchSize

```

## 2.2.47. cublasSgemmStridedBatched

SGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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,
      batchSize)
  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 :: batchSize

```

```

integer(4) function cublasSgemmStridedBatched_v2(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchSize)
  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 :: batchSize

```

## 2.3. Double Precision Functions and Subroutines

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

### 2.3.1. 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

```

## 2.3.2. 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
```

## 2.3.3. 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
```

## 2.3.4. 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
```

## 2.3.5. 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
```

## 2.3.6. 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
```

## 2.3.7. dnorm2

DNRM2 returns the euclidean norm of a vector via the function name, so that  $\text{DNRM2} := \sqrt{x^T x}$

```
real(8) function dnorm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x ! device or host variable
  integer :: incx
```

```
real(8) function cublasDnorm2(n, x, incx)
  integer :: n
  real(8), device, dimension(*) :: x
  integer :: incx
```

```
integer(4) function cublasDnorm2_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
```



## 2.3.8. 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
```

## 2.3.9. 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
```

## 2.3.10. 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
```

### 2.3.11. drotmg

DROTMG constructs the modified Givens transformation matrix H which zeros the second component of the 2-vector  $(\text{SQRT}(\text{DD1}) \cdot \text{DX1}, \text{SQRT}(\text{DD2}) \cdot \text{DY2})^T$ . With  $\text{DPARAM}(1) = \text{DFLAG}$ , H has one of the following forms..  $\text{DFLAG} = -1.00$   $\text{DFLAG} = 0.00$   $\text{DFLAG} = 1.00$   $\text{DFLAG} = -2.00$  (DH11 DH12) (1.00 DH12) (DH11 1.00) (1.00 0.00)  $H = \begin{pmatrix} ( ) & ( ) \\ ( ) & ( ) \end{pmatrix}$  (DH21 DH22), (DH21 1.00), (-1.00 DH22), (0.00 1.00). Locations 2-4 of  $\text{DPARAM}$  contain DH11, DH21, DH12, and DH22 respectively. (Values of 1.00, -1.00, or 0.00 implied by the value of  $\text{DPARAM}(1)$  are not stored in  $\text{DPARAM}$ .)

```
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
```

### 2.3.12. 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
```

### 2.3.13. 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
```

## 2.3.14. 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
```

## 2.3.15. 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
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x, y
  real(8), device :: alpha, beta ! device or host variable
```

## 2.3.16. 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
```

## 2.3.17. 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
  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
```

## 2.3.18. 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

```

### 2.3.19. 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
  integer :: t
  integer :: n, incx
  real(8), device, dimension(*) :: a, x
  real(8), device :: alpha ! device or host variable

```

### 2.3.20. 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

```

### 2.3.21. 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
```

### 2.3.22. 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
```

### 2.3.23. 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 cublasDsyrr2(t, n, alpha, x, incx, y, incy, a, lda)
character*1 :: 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
```

```
integer(4) function cublasDsyrr2_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
```

### 2.3.24. 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
```

### 2.3.25. 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

```

### 2.3.26. 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

```

### 2.3.27. 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

```

### 2.3.28. 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
```

### 2.3.29. 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
  integer :: u, t, d
  integer :: n, incx, lda
  real(8), device, dimension(lda, *) :: a
  real(8), device, dimension(*) :: x
```

### 2.3.30. dgemm

DGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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

```

### 2.3.31. 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

```

### 2.3.32. 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 cublasDsyrrk(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 cublasDsyrrk_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
```

### 2.3.33. 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 cublasDsyrrk2k(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 cublasDsyrrk2k_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
```

### 2.3.34. 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 cublasDsyrrkx(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 cublasDsyrrkx_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

```

### 2.3.35. dtrmm

DTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ .

```

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

```

### 2.3.36. dtrsm

DTRSM solves one of the matrix equations  $\text{op}(A) * X = \alpha * B$ , or  $X * \text{op}(A) = \alpha * B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ . 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
```

### 2.3.37. cublasDgemvBatched

DGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: xarray(*)
  integer :: incx
  real(8), device :: beta ! device or host variable
  type(c_devpstr), 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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: xarray(*)
  integer :: incx
  real(8), device :: beta ! device or host variable
  type(c_devpstr), device :: yarray(*)
  integer :: incy
  integer :: batchCount
```

### 2.3.38. cublasDgemmBatched

DGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Barray(*)
```

```

integer :: ldb
real(8), device :: beta ! device or host variable
type(c_devpstr), device :: Carray(*)
integer :: ldc
integer :: batchSize

```

```

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

```

### 2.3.39. 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  $m \geq n$ : 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  $m \geq n$ : 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, batchSize)
    type(cublasHandle) :: h
    integer :: trans ! integer or character(1) variable
    integer :: m, n, nrhs
    type(c_devpstr), device :: Aarray(*)
    integer :: lda
    type(c_devpstr), device :: Carray(*)
    integer :: ldc
    integer :: info(*)
    integer, device :: devinfo(*)
    integer :: batchSize

```

### 2.3.40. 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,
    batchSize)
    type(cublasHandle) :: h
    integer :: m, n
    type(c_devpstr), device :: Aarray(*)
    integer :: lda
    type(c_devpstr), device :: Tau(*)
    integer :: info(*)
    integer :: batchSize

```

### 2.3.41. 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
```

### 2.3.42. cublasDgetriBatched

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

```
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
```

### 2.3.43. 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
```

### 2.3.44. 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  type(c_devp_ptr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

### 2.3.45. cublasDtrsmBatched

DTRSM solves one of the matrix equations  $\text{op}(A)X = \alpha B$ , or  $X\text{op}(A) = \alpha B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ . 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
  type(c_devp_ptr), device :: A(*)
  integer :: lda
  type(c_devp_ptr), 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_devp_ptr), device :: A(*)
  integer :: lda
  type(c_devp_ptr), device :: B(*)
  integer :: ldb
  integer :: batchCount
```

### 2.3.46. cublasDgemvStridedBatched

DGEMV performs a batch of the matrix-vector operations  $Y := \alpha \text{op}(A) * X + \beta Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  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
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

```

### 2.3.47. cublasDgemmStridedBatched

DGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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 :: batchSize

```

## 2.4. Single Precision Complex Functions and Subroutines

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

### 2.4.1. 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

```

### 2.4.2. 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
```

### 2.4.3. 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
```

### 2.4.4. 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
```

### 2.4.5. 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
```

## 2.4.6. 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
```

## 2.4.7. 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
```

## 2.4.8. 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
```

## 2.4.9. 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
  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
```

## 2.4.10. 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
```

## 2.4.11. 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
```

```

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

```

## 2.4.12. 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

```

## 2.4.13. 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

```

## 2.4.14. 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
```

## 2.4.15. 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
```

## 2.4.16. 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
```

## 2.4.17. 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
```

## 2.4.18. 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
```

## 2.4.19. 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
```

## 2.4.20. 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
```

## 2.4.21. 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
```

## 2.4.22. 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
```

## 2.4.23. 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
```

## 2.4.24. 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
```

## 2.4.25. 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
```

## 2.4.26. 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
```

## 2.4.27. 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
```

## 2.4.28. 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
```

## 2.4.29. 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
  complex(4), device, dimension(*) :: x, y
  complex(4), device :: alpha, beta ! device or host variable

```

### 2.4.30. 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

```

### 2.4.31. 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
```

## 2.4.32. cher2

CHER2 performs the hermitian rank 2 operation  $A := \alpha x y^H + \text{conjg}(\alpha) y x^H + A$ , 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
```

## 2.4.33. 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
```

## 2.4.34. chpr2

CHPR2 performs the hermitian rank 2 operation  $A := \alpha x y^H + \text{conjg}(\alpha) y x^H + A$ , 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

```

### 2.4.35. cgemmm

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

```

subroutine cgemmm(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 cublasCgemmm(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 cublasCgemmm_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
  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

```

### 2.4.36. 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

```

### 2.4.37. 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 cublasCsyrrk(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 cublasCsyrrk_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

```

### 2.4.38. 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 cublasCsyrr2k(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 cublasCsyrr2k_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
```

## 2.4.39. 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 cublasCsyrrkx(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 cublasCsyrrkx_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
```

## 2.4.40. ctrmm

CTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$  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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ .

```
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

```

### 2.4.41. ctrsm

CTRSM solves one of the matrix equations  $\text{op}(A)X = \alpha B$ , or  $X\text{op}(A) = \alpha B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ . 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

```

### 2.4.42. 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 cublasChemv_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

```

### 2.4.43. 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

```

### 2.4.44. cher2k

CHER2K performs one of the hermitian rank 2k operations  $C := \alpha * A * B^* * H + \text{conjg}(\alpha) * B * A^* * H + \beta * C$ , or  $C := \alpha * A^* * H * B + \text{conjg}(\alpha) * B^* * H * A + \beta * C$ , 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

```

### 2.4.45. 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
  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

```

### 2.4.46. cublasCgemvBatched

CGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{*T}$ ,  $\alpha$  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, batchSize)
  type(cublasHandle) :: h
  integer :: trans ! integer or character(1) variable
  integer :: m, n
  complex(4), device :: alpha ! device or host variable
  type(c_devpnr), device :: Aarray(*)
  integer :: lda
  type(c_devpnr), device :: xarray(*)

```

```

integer :: incx
complex(4), device :: beta ! device or host variable
type(c_devpstr), device :: yarray(*)
integer :: incy
integer :: batchSize

```

```

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

```

## 2.4.47. cublasCgemmBatched

CGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$  or  $\text{op}(X) = X^{**H}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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, batchSize)
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_devpstr), device :: Aarray(*)
integer :: lda
type(c_devpstr), device :: Barray(*)
integer :: ldb
complex(4), device :: beta ! device or host variable
type(c_devpstr), device :: Carray(*)
integer :: ldc
integer :: batchSize

```

```

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

```

## 2.4.48. 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  $m \geq n$ : 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  $m \geq n$ : 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  type(c_devp_ptr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

### 2.4.49. 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  type(c_devp_ptr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

### 2.4.50. 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_devp_ptr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

## 2.4.51. cublasCgetriBatched

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

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

## 2.4.52. 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_devpnr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devpnr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

## 2.4.53. 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_devpnr), device :: Aarray(*)
  integer :: lda
  type(c_devpnr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

## 2.4.54. cublasCtrsmBatched

CTRSM solves one of the matrix equations  $\text{op}(A)*X = \alpha*B$ , or  $X*\text{op}(A) = \alpha*B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^*T$  or  $\text{op}(A) = A^*H$ . 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
```

## 2.4.55. cublasCgemvStridedBatched

CGEMV performs a batch of the matrix-vector operations  $Y := \alpha \cdot \text{op}(A) \cdot X + \beta \cdot Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^*T$ ,  $\alpha$  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 :: batchSize

```

## 2.4.56. cublasCgemvStridedBatched

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

```

integer(4) function cublasCgemvStridedBatched(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
  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 :: batchSize

```

```

integer(4) function cublasCgemvStridedBatched_v2(h, transa, transb, m, n, k,
alpha, Aarray, lda, strideA, Barray, ldb, strideB, beta, Carray, ldc,
strideC, batchSize)
  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 :: batchSize

```

## 2.5. Double Precision Complex Functions and Subroutines

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

## 2.5.1. 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
```

## 2.5.2. 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
```

## 2.5.3. 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
  complex(8), device, dimension(*) :: x
  integer :: incx
  real(8), device :: res ! device or host variable
```

## 2.5.4. 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
```

## 2.5.5. 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
```

## 2.5.6. 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
```

## 2.5.7. 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
```

## 2.5.8. 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
```

## 2.5.9. 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

```

## 2.5.10. 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

```

## 2.5.11. 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

```

## 2.5.12. 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
```

## 2.5.13. 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
```

## 2.5.14. 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
```

## 2.5.15. 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
```

## 2.5.16. 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
  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
```

## 2.5.17. 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
```

## 2.5.18. 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
```

## 2.5.19. 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
```

## 2.5.20. 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
```

## 2.5.21. 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
```

## 2.5.22. 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
  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

```

### 2.5.23. 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

```

### 2.5.24. 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

```

### 2.5.25. 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
```

## 2.5.26. 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
```

## 2.5.27. 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
```

## 2.5.28. 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
```

## 2.5.29. 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
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

## 2.5.30. 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
```

## 2.5.31. 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
```

## 2.5.32. zher2

ZHER2 performs the hermitian rank 2 operation  $A := \alpha * x * y^H + \text{conjg}(\alpha) * y * x^H + A$ , 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

```

### 2.5.33. 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

```

### 2.5.34. zhpr2

ZHPR2 performs the hermitian rank 2 operation  $A := \alpha * x * y^* H + \text{conjg}(\alpha) * y * x^* H + A$ , 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

```

## 2.5.35. zgemm

ZGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$  or  $\text{op}(X) = X^{**H}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
  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
```

## 2.5.36. 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
```

### 2.5.37. zsyrrk

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 zsyrrk(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 cublasZsyrrk(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 cublasZsyrrk_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
```

### 2.5.38. zsyrr2k

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 zsyrr2k(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 cublasZsyrr2k(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 cublasZsyrr2k_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
```



## 2.5.39. zsyrrkx

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 zsyrrkx(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 cublasZsyrrkx(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 cublasZsyrrkx_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
```

## 2.5.40. ztrmm

ZTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$  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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A * T$  or  $\text{op}(A) = A * H$ .

```
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
```

## 2.5.41. ztrsm

ZTRSM solves one of the matrix equations  $\text{op}(A)X = \alpha B$ , or  $X\text{op}(A) = \alpha B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^*T$  or  $\text{op}(A) = A^*H$ . 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
```

## 2.5.42. 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
```

### 2.5.43. 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
```

### 2.5.44. zher2k

ZHER2K performs one of the hermitian rank 2k operations  $C := \alpha * A * B^{*H} + \text{conj}(\alpha) * B * A^{*H} + \beta * C$ , or  $C := \alpha * A^{*H} * B + \text{conj}(\alpha) * B^{*H} * A + \beta * C$ , 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

```

## 2.5.45. 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
  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

```

## 2.5.46. cublasZgemvBatched

ZGEMV performs a batch of the matrix-vector operations  $Y := \alpha \text{op}(A) * X + \beta Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^* T$ ,  $\alpha$  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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: xarray(*)
  integer :: incx
  complex(8), device :: beta ! device or host variable
  type(c_devpstr), 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_devpnr), device :: Aarray(*)
integer :: lda
type(c_devpnr), device :: xarray(*)
integer :: incx
complex(8), device :: beta ! device or host variable
type(c_devpnr), device :: yarray(*)
integer :: incy
integer :: batchCount

```

## 2.5.47. cublasZgemvBatched

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

```

integer(4) function cublasZgemvBatched(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_devpnr), device :: Aarray(*)
  integer :: lda
  type(c_devpnr), device :: Barray(*)
  integer :: ldb
  complex(8), device :: beta ! device or host variable
  type(c_devpnr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount

```

```

integer(4) function cublasZgemvBatched_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_devpnr), device :: Aarray(*)
  integer :: lda
  type(c_devpnr), device :: Barray(*)
  integer :: ldb
  complex(8), device :: beta ! device or host variable
  type(c_devpnr), device :: Carray(*)
  integer :: ldc
  integer :: batchCount

```

## 2.5.48. 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  $\text{TRANS} = 'N'$  and  $m \geq n$ : find the least squares solution of an overdetermined system, i.e., solve the least squares problem minimize  $\|B - A * X\|$ . 2. If  $\text{TRANS} = 'N'$  and  $m < n$ : find the minimum norm solution of an underdetermined system  $A * X = B$ . 3. If  $\text{TRANS} = 'C'$  and  $m \geq n$ : find the minimum norm solution of an underdetermined system  $A^{**H} * X = B$ . 4. If  $\text{TRANS} = 'C'$  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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Carray(*)
  integer :: ldc
  integer :: info(*)
  integer, device :: devinfo(*)
  integer :: batchCount
```

## 2.5.49. 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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Tau(*)
  integer :: info(*)
  integer :: batchCount
```

## 2.5.50. 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_devpstr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  integer, device :: info(*)
  integer :: batchCount
```

## 2.5.51. cublasZgetriBatched

ZGETRI computes the inverse of a matrix using the LU factorization computed by ZGETRF. This method inverts  $U$  and then computes  $\text{inv}(A)$  by solving the system  $\text{inv}(A)*L = \text{inv}(U)$  for  $\text{inv}(A)$ .

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

```
integer :: ldc
integer, device :: info(*)
integer :: batchCount
```

## 2.5.52. 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_devpstr), device :: Aarray(*)
  integer :: lda
  integer, device :: ipvt(*)
  type(c_devpstr), device :: Barray(*)
  integer :: ldb
  integer :: info(*)
  integer :: batchCount
```

## 2.5.53. 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_devpstr), device :: Aarray(*)
  integer :: lda
  type(c_devpstr), device :: Ainv(*)
  integer :: lda_inv
  integer, device :: info(*)
  integer :: batchCount
```

## 2.5.54. 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_devpstr), device :: A(*)
  integer :: lda
  type(c_devpstr), 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 :: batchSize

```

## 2.5.55. cublasZgemvStridedBatched

ZGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  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, batchSize)
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 :: batchSize

```

```

integer(4) function cublasZgemvStridedBatched_v2(h, trans, m, n, alpha, &
    A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchSize)
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 :: batchSize

```

## 2.5.56. cublasZgemmStridedBatched

ZGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  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
  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
```

## 2.6. 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.

## 2.6.1. cublasHgemvBatched

HGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^*T$ ,  $\alpha$  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_devpstr), device :: Aarray(*)
    integer :: lda
    type(c_devpstr), device :: xarray(*)
    integer :: incx
    real(4), device :: beta ! device or host variable
    type(c_devpstr), 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_devpstr), device :: Aarray(*)
    integer :: lda
    type(c_devpstr), device :: xarray(*)
    integer :: incx
    real(4), device :: beta ! device or host variable
    type(c_devpstr), 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_devpstr), device :: Aarray(*)
    integer :: lda
    type(c_devpstr), device :: xarray(*)
    integer :: incx
    real(4), device :: beta ! device or host variable
    type(c_devpstr), 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_devpstr), 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 :: batchSize

```

## 2.6.2. cublasHgemvStridedBatched

HGEMV performs a batch of the matrix-vector operations  $Y := \alpha * \text{op}(A) * X + \beta * Y$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{*T}$ ,  $\alpha$  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, batchSize)
    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 :: batchSize

```

```

integer(4) function cublasHSHgemvStridedBatched_v2(h, trans, m, n, alpha, &
    A, lda, strideA, X, incx, strideX, beta, Y, incy, strideY, batchSize)
    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(2), device :: Y(*)
    integer :: incy
    integer(8) :: strideY
    integer :: batchSize

```

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, batchSize)
    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

```

### 2.6.3. cublasHgemm

HGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
  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

```

### 2.6.4. cublasHgemmBatched

HGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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 :: batchSize

```

```

integer(4) function cublasHgemmBatched_v2(h, transa, transb, m, n, k, &
    alpha, Aarray, lda, Barray, ldb, beta, Carray, ldc, batchSize)
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
type(c_devptr), device :: Carray(*)
integer :: ldc
integer :: batchSize

```

## 2.6.5. cublasHgemmStridedBatched

HGEMM performs a set of matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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, batchSize)
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 :: batchSize

```

```

integer(4) function cublasHgemmStridedBatched_v2(h, transa, transb, m, n, k, &
    alpha, A, lda, strideA, B, ldb, strideB, beta, C, ldc, strideC, batchSize)
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 :: batchSize

```

## 2.6.6. 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

```

## 2.6.7. 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

```

## 2.6.8. 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

```

## 2.6.9. cublasAxyEx

AXPY computes a constant times a vector plus a vector.

```

integer(4) function cublasAxyEx(h, n, alpha, alphatype, &
  x, xtype, incx, y, ytype, incy, extype)
  type(cublasHandle) :: h
  integer :: n
  real(2), device :: alpha
  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) :: exctype
```

## 2.6.10. 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
```

## 2.6.11. cublasDotEx

**DOT** forms the dot product of two vectors.

```
integer(4) function cublasDotEx(h, n, x, xtype, incx, &
    y, ytype, incy, res, restype, exctype)
    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) :: exctype
```

## 2.6.12. cublasDotcEx

**DOTC** forms the conjugated dot product of two vectors.

```
integer(4) function cublasDotcEx(h, n, x, xtype, incx, &
    y, ytype, incy, res, restype, exctype)
    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) :: exctype
```

## 2.6.13. cublasNrm2Ex

**NRM2** produces the euclidean norm of a vector.

```
integer(4) function cublasNrm2Ex(h, n, x, xtype, incx, res, &
    restype, exctype)
    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

```

## 2.6.14. 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

```

## 2.6.15. 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

```

## 2.6.16. 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

```

## 2.6.17. cublasRotmgEx

ROTMG constructs a modified Givens transformation matrix.

```

integer(4) function cublasRotmgEx(h, d1, dltype, d2, d2type, &
    x1, xltype, y1, yltype, param, paramtype, extype)
    type(cublasHandle) :: h
    real(2), device :: d1 ! Type and kind as specified by dltype
    type(cudaDataType) :: dltype
    real(2), device :: d2 ! Type and kind as specified by d2type
    type(cudaDataType) :: d2type

```



```

real(2), device :: x1 ! Type and kind as specified by xtype
type(cudaDataType) :: xtype
real(2), device :: y1 ! Type and kind as specified by ydtype
type(cudaDataType) :: ydtype
real(2), device, dimension(*) :: param
type(cudaDataType) :: paramtype
type(cudaDataType) :: exdtype

```

## 2.6.18. cublasScalEx

SCAL scales a vector by a constant.

```

integer(4) function cublasScalEx(h, n, alpha, alphas, &
    x, xtype, incx, exdtype)
    type(cublasHandle) :: h
    integer :: n
    real(2), device :: alpha
    type(cudaDataType) :: alphas
    real(2), device, dimension(*) :: x
    type(cudaDataType) :: xtype
    integer :: incx
    type(cudaDataType) :: exdtype

```

## 2.6.19. cublasSwapEx

SWAP interchanges two vectors.

```

integer(4) function cublasSwapEx(h, n, x, xtype, incx, &
    y, ydtype, incy)
    type(cublasHandle) :: h
    integer :: n
    real(2), device, dimension(*) :: x
    type(cudaDataType) :: xtype
    integer :: incx
    real(2), device, dimension(*) :: y
    type(cudaDataType) :: ydtype
    integer :: incy

```

## 2.6.20. cublasGemmEx

GEMM performs the matrix-matrix multiply operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
    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
```

## 2.6.21. cublasGemmBatchedEx

GEMM performs a batch of matrix-matrix multiply operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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, batchSize, computeType, algo)
type(cublasHandle) :: h
integer :: transa, transb
integer :: m, n, k
real(2), device :: alpha ! device or host variable
type(c_devp_ptr), device :: Aarray(*)
type(cudaDataType) :: atype
integer :: lda
type(c_devp_ptr), device :: Barray(*)
type(cudaDataType) :: btype
integer :: ldb
real(2), device :: beta ! device or host variable
type(c_devp_ptr), device :: Carray(*)
type(cudaDataType) :: ctype
integer :: ldc
integer :: batchSize
type(cublasComputeType) :: computeType ! also accept integer
type(cublasGemmAlgoType) :: algo ! also accept integer
```

## 2.6.22. cublasGemmStridedBatchedEx

GEMM performs a batch of matrix-matrix multiply operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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, batchSize, 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
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 :: batchSize
type(cublasComputeType) :: computeType ! also accept integer
type(cublasGemmAlgoType) :: algo ! also accept integer
```

## 2.7. 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.

### 2.7.1. Single Precision Functions and Subroutines

This section contains the V2 interfaces to the single precision BLAS and cuBLAS functions and subroutines.

#### 2.7.1.1. 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
```

#### 2.7.1.2. 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
```

#### 2.7.1.3. 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
```

#### 2.7.1.4. 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
```

#### 2.7.1.5. 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
```

#### 2.7.1.6. 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
```

#### 2.7.1.7. 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
```

#### 2.7.1.8. 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
```

### 2.7.1.9. 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
```

### 2.7.1.10. 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
```

### 2.7.1.11. 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
```

### 2.7.1.12. 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
```

### 2.7.1.13. 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
```

### 2.7.1.14. 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

```

### 2.7.1.15. 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

```

### 2.7.1.16. 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

```

### 2.7.1.17. 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

```

### 2.7.1.18. 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

```

### 2.7.1.19. 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
```

### 2.7.1.20. 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
```

### 2.7.1.21. 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
```

### 2.7.1.22. 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
```

### 2.7.1.23. 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
```

### 2.7.1.24. 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
```

### 2.7.1.25. 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
```

### 2.7.1.26. 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
```

### 2.7.1.27. 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
```

### 2.7.1.28. 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
```

### 2.7.1.29. 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
```



### 2.7.1.30. 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
```

### 2.7.1.31. 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
```

### 2.7.1.32. 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
```

### 2.7.1.33. 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
```

### 2.7.1.34. 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
```

### 2.7.1.35. 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
```

### 2.7.1.36. 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
```

## 2.7.2. Double Precision Functions and Subroutines

This section contains the V2 interfaces to the double precision BLAS and cuBLAS functions and subroutines.

### 2.7.2.1. 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
```

### 2.7.2.2. 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
```

### 2.7.2.3. 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
```

### 2.7.2.4. 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
```

### 2.7.2.5. 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
```

### 2.7.2.6. 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
```

### 2.7.2.7. dnrnm2

If you use the `cublas_v2` module, the interface for `cublasDnrnm2` is changed to the following:

```
integer(4) function cublasDnrnm2(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
```

### 2.7.2.8. 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
```

### 2.7.2.9. 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
```

### 2.7.2.10. 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
```

### 2.7.2.11. 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
```

### 2.7.2.12. 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
```

### 2.7.2.13. 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
```

### 2.7.2.14. 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
```

### 2.7.2.15. 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
```

### 2.7.2.16. 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
```

### 2.7.2.17. 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

```

### 2.7.2.18. 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

```

### 2.7.2.19. 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

```

### 2.7.2.20. 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

```

### 2.7.2.21. 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

```

### 2.7.2.22. 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

```

### 2.7.2.23. 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

```

### 2.7.2.24. 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

```

### 2.7.2.25. 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

```

### 2.7.2.26. 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

```

### 2.7.2.27. 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
```

### 2.7.2.28. 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
```

### 2.7.2.29. 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
```

### 2.7.2.30. 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
```

### 2.7.2.31. 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
```



### 2.7.2.32. dsyrk

If you use the `cublas_v2` module, the interface for `cublasDsyrc` is changed to the following:

```
integer(4) function cublasDsyrc(h, uplo, trans, n, k, alpha, a, lda, beta, c,
lda)
  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
```

### 2.7.2.33. dsyr2k

If you use the `cublas_v2` module, the interface for `cublasDsyrc2k` is changed to the following:

```
integer(4) function cublasDsyrc2k(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
```

### 2.7.2.34. dsyrkx

If you use the `cublas_v2` module, the interface for `cublasDsyrcx` is changed to the following:

```
integer(4) function cublasDsyrcx(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
```

### 2.7.2.35. 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
```

### 2.7.2.36. 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
```

## 2.7.3. Single Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the single precision complex BLAS and cuBLAS functions and subroutines.

### 2.7.3.1. 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
```

### 2.7.3.2. 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
```

### 2.7.3.3. 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
```

### 2.7.3.4. 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
```

### 2.7.3.5. 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
```

### 2.7.3.6. 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
```

### 2.7.3.7. 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
```

### 2.7.3.8. 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
```

### 2.7.3.9. 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
```

### 2.7.3.10. 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
```

### 2.7.3.11. 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
```

### 2.7.3.12. 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
```

### 2.7.3.13. 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
```

### 2.7.3.14. 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
```

### 2.7.3.15. 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
```

### 2.7.3.16. 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
```

### 2.7.3.17. 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
```

### 2.7.3.18. 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
```

### 2.7.3.19. 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
```

### 2.7.3.20. 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
```

### 2.7.3.21. 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
```

### 2.7.3.22. 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
```

### 2.7.3.23. 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
```

### 2.7.3.24. 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
```

### 2.7.3.25. 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
```

### 2.7.3.26. 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
```

### 2.7.3.27. 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
```

### 2.7.3.28. 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
```

### 2.7.3.29. 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
```

### 2.7.3.30. 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
```

### 2.7.3.31. 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
```

### 2.7.3.32. 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
```

### 2.7.3.33. 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
```



### 2.7.3.34. 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
```

### 2.7.3.35. 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
```

### 2.7.3.36. 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
```

### 2.7.3.37. 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
```

### 2.7.3.38. 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
```

### 2.7.3.39. 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
```

### 2.7.3.40. 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
```

### 2.7.3.41. 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
  complex(4), device, dimension(ldb, *) :: b
  complex(4), device :: alpha ! device or host variable
```

### 2.7.3.42. 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
```

### 2.7.3.43. 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
```

### 2.7.3.44. 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
```

### 2.7.3.45. 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
```

## 2.7.4. Double Precision Complex Functions and Subroutines

This section contains the V2 interfaces to the double precision complex BLAS and cuBLAS functions and subroutines.

### 2.7.4.1. 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
```

### 2.7.4.2. 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
```

### 2.7.4.3. 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
```

### 2.7.4.4. 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
```

### 2.7.4.5. 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
```

### 2.7.4.6. 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
```

### 2.7.4.7. 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
```

### 2.7.4.8. dznorm2

If you use the `cublas_v2` module, the interface for `cublasDznorm2` is changed to the following:

```
integer(4) function cublasDznorm2(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
```

### 2.7.4.9. 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
```

### 2.7.4.10. 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
```

### 2.7.4.11. 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
```

### 2.7.4.12. 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
```

### 2.7.4.13. 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
```

### 2.7.4.14. 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
```

### 2.7.4.15. 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
  complex(8), device, dimension(lda, *) :: a
  complex(8), device, dimension(*) :: x, y
  complex(8), device :: alpha, beta ! device or host variable
```

### 2.7.4.16. 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
```

### 2.7.4.17. 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
```

### 2.7.4.18. 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
```

### 2.7.4.19. 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

```

### 2.7.4.20. zsyrr

If you use the `cublas_v2` module, the interface for `cublasZsyrr` is changed to the following:

```

integer(4) function cublasZsyrr(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

```

### 2.7.4.21. zsyrr2

If you use the `cublas_v2` module, the interface for `cublasZsyrr2` is changed to the following:

```

integer(4) function cublasZsyrr2(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

```

### 2.7.4.22. 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

```

### 2.7.4.23. 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

```

### 2.7.4.24. 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
```

### 2.7.4.25. 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
```

### 2.7.4.26. 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
```

### 2.7.4.27. 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
```

### 2.7.4.28. zhbmvm

If you use the `cublas_v2` module, the interface for `cublasZhbmvm` is changed to the following:

```
integer(4) function cublasZhbmvm(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
```

### 2.7.4.29. 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
```

### 2.7.4.30. 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
```

### 2.7.4.31. 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
```

### 2.7.4.32. 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
```

### 2.7.4.33. 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
```

### 2.7.4.34. 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
```

### 2.7.4.35. 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
```

### 2.7.4.36. 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
```

### 2.7.4.37. 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
```

### 2.7.4.38. 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
```

### 2.7.4.39. zsyrrkx

If you use the `cublas_v2` module, the interface for `cublasZsyrrkx` is changed to the following:

```
integer(4) function cublasZsyrrkx(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
```

### 2.7.4.40. 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
```

### 2.7.4.41. 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
```

### 2.7.4.42. 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
```

### 2.7.4.43. 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
```

### 2.7.4.44. 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
```

### 2.7.4.45. 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
```

## 2.8. 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, &
                    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

```

### 2.8.1. 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

```

## 2.8.2. 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
```

## 2.8.3. 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
```

## 2.8.4. 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
```

## 2.8.5. 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
```

## 2.8.6. 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
```

### 2.8.7. 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
```

### 2.8.8. 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 CUDA 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
```

### 2.8.9. 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
```

### 2.8.10. cublasXtSgemm

SGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
```



## 2.8.11. 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_intp_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
```

## 2.8.12. cublasXtSsyk

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_intp_t) :: n, k, lda, ldc
  real(4), dimension(lda, *) :: a
  real(4), dimension(ldc, *) :: c
  real(4) :: alpha, beta
```

## 2.8.13. cublasXtSsy2k

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_intp_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
```

## 2.8.14. cublasXtSsykx

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

```

## 2.8.15. cublasXtStrmm

STRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ .

```

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

```

## 2.8.16. cublasXtStrsm

STRSM solves one of the matrix equations  $\text{op}(A) * X = \alpha * B$ , or  $X * \text{op}(A) = \alpha * B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ . 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

```

## 2.8.17. 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

```

## 2.8.18. cublasXtCgemm

CGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$  or  $\text{op}(X) = X^{**H}$ ,  $\alpha$  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
```

## 2.8.19. cublasXtChemmm

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
```

## 2.8.20. 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
```

## 2.8.21. cublasXtCher2k

CHER2K performs one of the hermitian rank  $2k$  operations  $C := \alpha * A * B^H + \text{conj}(\alpha) * B * A^H + \beta * C$ , or  $C := \alpha * A^H * B + \text{conj}(\alpha) * B^H * A + \beta * C$ , 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
```

## 2.8.22. 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
```

## 2.8.23. 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
```

## 2.8.24. 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
```

## 2.8.25. 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

```

## 2.8.26. cublasXtCsyrrkx

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 cublasXtcsyrrkx(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

```

## 2.8.27. cublasXtCtrmm

CTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$  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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ .

```

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

```

## 2.8.28. cublasXtCtrsm

CTRSM solves one of the matrix equations  $\text{op}(A) * X = \alpha * B$ , or  $X * \text{op}(A) = \alpha * B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ . 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

```

## 2.8.29. 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 cublasXtcsppmm(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
```

## 2.8.30. cublasXtDgemm

DGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
```

## 2.8.31. cublasXtDsymb

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 cublasXtdsymb(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
```

## 2.8.32. 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

```

### 2.8.33. cublasXtDsyrr2k

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 cublasXtdsyrr2k(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

```

### 2.8.34. cublasXtDsyrrkx

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 cublasXtdsyrrkx(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

```

### 2.8.35. cublasXtDtrmm

DTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$ , 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  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ .

```

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

```

### 2.8.36. cublasXtDtrsm

DTRSM solves one of the matrix equations  $\text{op}(A) * X = \alpha * B$ , or  $X * \text{op}(A) = \alpha * B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^*T$ . The matrix  $X$  is overwritten on  $B$ .

```
integer(4) function cublasXttrsm(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
```

## 2.8.37. 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
```

## 2.8.38. cublasXtZgemm

ZGEMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^*T$  or  $\text{op}(X) = X^*H$ ,  $\alpha$  and  $\beta$  are scalars, and  $A$ ,  $B$  and  $C$  are matrices, with  $\text{op}(A)$  an  $m$  by  $k$  matrix,  $\text{op}(B)$  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
```

## 2.8.39. 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
```



## 2.8.40. 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_intp_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  real(8) :: alpha, beta
```

## 2.8.41. cublasXtZher2k

ZHER2K performs one of the hermitian rank  $2k$  operations  $C := \alpha * A * B^{*H} + \text{conjg}(\alpha) * B * A^{*H} + \beta * C$ , or  $C := \alpha * A^{*H} * B + \text{conjg}(\alpha) * B^{*H} * A + \beta * C$ , 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_intp_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
```

## 2.8.42. 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_intp_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
```

### 2.8.43. 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_intp_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
```

### 2.8.44. 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_intp_t) :: n, k, lda, ldc
  complex(8), dimension(lda, *) :: a
  complex(8), dimension(ldc, *) :: c
  complex(8) :: alpha, beta
```

### 2.8.45. 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_intp_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
```

### 2.8.46. 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

```

### 2.8.47. cublasXtZtrmm

ZTRMM performs one of the matrix-matrix operations  $B := \alpha * \text{op}(A) * B$ , or  $B := \alpha * B * \text{op}(A)$  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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ .

```

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

```

### 2.8.48. cublasXtZtrsm

ZTRSM solves one of the matrix equations  $\text{op}(A) * X = \alpha * B$ , or  $X * \text{op}(A) = \alpha * B$ , 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  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$  or  $\text{op}(A) = A^{**H}$ . 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

```

### 2.8.49. 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

```

# Chapter 3.

## 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).

### 3.1. 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
enumerator :: CUFFT_EXEC_FAILED    = 6
enumerator :: CUFFT_SETUP_FAILED   = 7
enumerator :: CUFFT_INVALID_SIZE   = 8
enumerator :: CUFFT_UNALIGNED_DATA = 9
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_NATIVE           = 0
enumerator :: CUFFT_COMPATIBILITY_FFTW_PADDING    = 1
enumerator :: CUFFT_COMPATIBILITY_FFTW_ASYMMETRIC = 2
enumerator :: CUFFT_COMPATIBILITY_FFTW_ALL        = 3
end enum

integer, parameter :: CUFFT_COMPATIBILITY_DEFAULT =
  CUFFT_COMPATIBILITY_FFTW_PADDING

```

### 3.1.1. cufftSetCompatibilityMode

This function configures the layout of cuFFT output in FFTW-compatible modes.

```

integer(4) function cufftSetCompatibilityMode( plan, mode )
  integer :: plan
  integer :: mode

```

### 3.1.2. 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

```

### 3.1.3. cufftGetVersion

This function returns the version number of cuFFT.

```

integer(4) function cufftGetVersion( version )
  integer :: version

```

### 3.1.4. 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

```

### 3.1.5. 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)
```

### 3.1.6. 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
```

## 3.2. CUFFT Plans and Estimated Size Functions

This section contains functions from the cuFFT library used to create plans and estimate work buffer size.

### 3.2.1. 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
```

### 3.2.2. 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
```

### 3.2.3. 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
```

### 3.2.4. 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
```

### 3.2.5. 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
```

### 3.2.6. 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(*)
```

### 3.2.7. 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(*)
```

### 3.2.8. 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(*)
```

### 3.2.9. 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)
  integer(4) :: istride, idist, ostride, odist
  integer(4) :: ffttype, batch
  integer(kind=int_ptr_kind()) :: workSize(*)
```

### 3.2.10. 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(*)
```

### 3.2.11. 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(*)
```

### 3.2.12. 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(*)
```

### 3.2.13. 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(*)
```

### 3.2.14. 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(*)
```

### 3.2.15. 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(*)
```

### 3.2.16. 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(*)
```

### 3.2.17. 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(*)
```

### 3.2.18. 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(*)
```

## 3.3. CUFFT Execution Functions

This section contains the execution functions, which perform the actual Fourier transform, in the cuFFT library.

### 3.3.1. 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
```

### 3.3.2. cufftExecR2C

This function executes a single precision real-to-complex, implicit 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
```

### 3.3.3. cufftExecC2R

This function executes a single precision complex-to-real, implicit 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
```

### 3.3.4. 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
```

### 3.3.5. cufftExecD2Z

This function executes a double precision real-to-complex, implicit 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
```

### 3.3.6. cufftExecZ2D

This function executes a double precision complex-to-real, implicit 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
```

## 3.4. 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_NVSHMEM     = 1
  enumerator :: CUFFT_COMM_UNDEFINED = 2
end enum

```

The `cufftXt` module contains the following derived type definitions:

```

! cufftXtldFactors type
type, bind(c) :: cufftXtldFactors
  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 cufftXtldFactors

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

```

### 3.4.1. 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(*)

```

### 3.4.2. 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
```

### 3.4.3. 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.
```

### 3.4.4. 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
```

## 3.5. CUFFTXT Plans and Work Area Functions

This section contains functions from the cufftXt library used to create plans and manage work buffers.

### 3.5.1. 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(*)
```

### 3.5.2. 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
```

### 3.5.3. 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
```

### 3.5.4. 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(*)
```

### 3.5.5. 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_devpstr) :: workArea(*)
```

### 3.5.6. 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
```

## 3.6. CUFFTXT Execution Functions

This section contains the execution functions, which perform the actual Fourier transform, in the `cufftXt` library.

### 3.6.1. 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
```

### 3.6.2. 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
```



### 3.6.3. 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
```

### 3.6.4. 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
```

### 3.6.5. 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
```

### 3.6.6. 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
```

### 3.6.7. 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
```

### 3.6.8. 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
```

## 3.7. CUFFTMP Functions

This section contains the cuFFTMp functions which extend the cuFFTXt library functionality to multiple processes and multiple GPUs.

### 3.7.1. cufftMpNvshmemMalloc

This function allocates space from the NVSHMEM symmetric heap. The cuFFTMp library is based on NVSHMEM. However, the user is not allowed to link and use NVSHMEM in their own application. This may cause a crash at application start time. This limitation will be lifted in a future release of cuFFTMp.

However, some functionality of cuFFTMp requires NVSHMEM-allocated memory, so this function is currently exposed and supported. This function requires that at least one cuFFTMp plan is active prior to its use.

```
integer(4) function cufftMpNvshmemMalloc( size, workArea )
  integer(8) :: size ! Size is in bytes
  type(c_devptr) :: workArea
```

### 3.7.2. cufftMpNvshmemFree

This function frees the space previously allocated from the NVSHMEM symmetric heap. The cuFFTMp library is based on NVSHMEM. However, the user is not allowed to link and use NVSHMEM in their own application. This may cause a crash at application start time. This limitation will be lifted in a future release of cuFFTMp.

However, some functionality of cuFFTMp requires NVSHMEM-allocated memory, so this function is currently exposed and supported. This function requires that at least one cuFFTMp plan is active prior to its use.

```
integer(4) function cufftMpNvshmemFree( workArea )
  type(c_devptr) :: workArea
```

### 3.7.3. 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
```

### 3.7.4. 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
```

### 3.7.5. cufftMpAttachReshapeComm

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
```

### 3.7.6. 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
```

### 3.7.7. cufftMpMakeReshape

This function creates a cuFFTMp reshape plan based on the input and output boxes. Note that the boxes use C conventions for bounds and strides.

```
integer(4) function cufftMpMakeReshape( reshapeHandle, &
  elementSize, boxIn, boxOut )
  type(c_ptr) :: reshapeHandle
  integer(8) :: elementSize
  type(cufftBox3d) :: boxIn
  type(cufftBox3d) :: boxOut
```

### 3.7.8. 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
```

### 3.7.9. cufftMpDestroyReshape

This function destroys a cuFFTMp reshape handle.

```
integer(4) function cufftMpDestroyReshape( reshapeHandle )
  type(c_ptr) :: reshapeHandle
```

# Chapter 4. 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)`.

## 4.1. 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
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
enumerator :: CURAND_3RD = 11
enumerator :: CURAND_DEFINITION = 12
enumerator :: CURAND_POISSON = 13
end enum

```

### 4.1.1. 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
```

### 4.1.2. 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
```

### 4.1.3. curandDestroyGenerator

This function destroys an existing random number generator.

```
integer(4) function curandDestroyGenerator(generator)
  type(curandGenerator) :: generator
```

### 4.1.4. curandGetVersion

This function returns the version number of the cuRAND library.

```
integer(4) function curandGetVersion(version)
  integer(4) :: version
```

### 4.1.5. 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
```

### 4.1.6. 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
```

### 4.1.7. 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
```

### 4.1.8. 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
```

### 4.1.9. curandSetQuasiRandomGeneratorDimensions

This function sets number of dimensions of the quasirandom number generator.

```
integer(4) function curandSetQuasiRandomGeneratorDimensions(generator, num)
  type(curandGenerator) :: generator
  integer(4) :: num
```

## 4.2. CURAND Generator Functions

This section contains interfaces for the cuRAND generator functions.

### 4.2.1. 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
```

### 4.2.2. 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
```

### 4.2.3. 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
```

#### 4.2.4. 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
```

#### 4.2.5. 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
```

#### 4.2.6. 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
```

#### 4.2.7. 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
```

#### 4.2.8. curandGenerateSeeds

This function sets the starting state of the generator.

```
integer(4) function curandGenerateSeeds(generator)
  type(curandGenerator) :: generator
```

#### 4.2.9. 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

```

## 4.2.10. 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

```

## 4.3. 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
  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 curandStateMtg32
    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 curandStateMtg32

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

```

### 4.3.1. 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.

#### 4.3.1.1. 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

```

#### 4.3.1.2. 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

```

#### 4.3.1.3. 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
```

#### 4.3.1.4. 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
```

#### 4.3.1.5. 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
```

#### 4.3.1.6. 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
```

#### 4.3.1.7. 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
```

## 4.3.2. 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.

### 4.3.2.1. 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
```

### 4.3.2.2. 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
```

### 4.3.2.3. 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
```

### 4.3.2.4. 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
```

### 4.3.2.5. 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
```

#### 4.3.2.6. 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
```

#### 4.3.2.7. 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
```

### 4.3.3. 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.

#### 4.3.3.1. 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
```

#### 4.3.3.2. 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
```

#### 4.3.3.3. 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
```

#### 4.3.3.4. 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
```

#### 4.3.3.5. 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
```

#### 4.3.3.6. 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
```

#### 4.3.3.7. 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
```

### 4.3.4. 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.

#### 4.3.4.1. 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
```

#### 4.3.4.2. 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
```

#### 4.3.4.3. 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
```

#### 4.3.4.4. 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
```

#### 4.3.4.5. 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
```

#### 4.3.4.6. 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
```

#### 4.3.4.7. 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
```

### 4.3.5. 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.

#### 4.3.5.1. 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
```

#### 4.3.5.2. 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
```

#### 4.3.5.3. 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
```

#### 4.3.5.4. 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
```

#### 4.3.5.5. curandLogNormalScrambledSobol32

This function returns a 32-bit floating point log-normally distributed random number from a 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
```

#### 4.3.5.6. 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
```

#### 4.3.5.7. curandLogNormalScrambledSobol64

This function returns a 32-bit floating point log-normally distributed random number from a 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
```

### 4.3.6. 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.

#### 4.3.6.1. 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
```

#### 4.3.6.2. 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
```

#### 4.3.6.3. 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
```

#### 4.3.6.4. 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
```

#### 4.3.6.5. 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
```

#### 4.3.6.6. 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
```

#### 4.3.6.7. 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
```

### 4.3.7. 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.

#### 4.3.7.1. 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
```

#### 4.3.7.2. 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
```

#### 4.3.7.3. 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
```

#### 4.3.7.4. 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
```

#### 4.3.7.5. 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
```

#### 4.3.7.6. 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
```

#### 4.3.7.7. curandUniformScrambledSobol64

This function returns a 32-bit floating point uniformly distributed random number from a 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
```

### 4.3.8. 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.

#### 4.3.8.1. 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
```

#### 4.3.8.2. 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
```

#### 4.3.8.3. 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
```

#### 4.3.8.4. 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
```

#### 4.3.8.5. 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
```

#### 4.3.8.6. 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
```

#### 4.3.8.7. 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 5.

## 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).

### 5.1. 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 is deprecated in CUDA 10.2, removed in CUDA 11.0
type cusparseSolveAnalysisInfo
  type(c_ptr) :: info
```

```

end type cusparseSolveAnalysisInfo

! This type is deprecated in CUDA 10.2, removed in CUDA 11.0
type cusparseHybMat
  type(c_ptr) :: mat
end type cusparseHybMat

type cusparseCsrsv2Info
  type(c_ptr) :: info
end type cusparseCsrsv2Info

type cusparseCsrlic02Info
  type(c_ptr) :: info
end type cusparseCsrlic02Info

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 cusparseBsrs2Info
  type(c_ptr) :: info
end type cusparseBsrs2Info

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 enumerations:

```

! 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_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
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
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
  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) ! cusparseSpSMAlg_t;
  enumerator :: CUSPARSE_SPSM_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_NONDETERMINISTIC = 2
end enum

enum, bind(C) ! cusparseSDDMMAlg_t;
  enumerator :: CUSPARSE_SDDMM_ALG_DEFAULT = 0
end enum

```

### 5.1.1. 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

```

### 5.1.2. 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

```

### 5.1.3. cusparseGetErrorName

This function returns the error code name.

```

character*128 function cusparseGetErrorName(ierr)
  integer(c_int) :: ierr

```

### 5.1.4. cusparseGetErrorString

This function returns the description string for an error code.

```

character*128 function cusparseGetErrorString(ierr)

```

```
integer(c_int) :: ierr
```

### 5.1.5. cusparseGetVersion

This function returns the version number of the cuSPARSE library.

```
integer(4) function cusparseGetVersion(handle, version)
  type(cusparseHandle) :: handle
  integer(c_int) :: version
```

### 5.1.6. 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
```

### 5.1.7. 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
```

### 5.1.8. 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
```

### 5.1.9. 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
```

### 5.1.10. 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
```

### 5.1.11. `cusparseDestroyMatDescr`

This function releases the memory allocated for the matrix descriptor.

```
integer(4) function cusparseDestroyMatDescr(descrA)
  type(cusparseMatDescr) :: descrA
```

### 5.1.12. `cusparseSetMatType`

This function sets the `MatrixType` of the matrix descriptor `descrA`.

```
integer(4) function cusparseSetMatType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

### 5.1.13. `cusparseGetMatType`

This function returns the `MatrixType` of the matrix descriptor `descrA`.

```
integer(4) function cusparseGetMatType(descrA)
  type(cusparseMatDescr) :: descrA
```

### 5.1.14. `cusparseSetMatFillMode`

This function sets the `FillMode` field of the matrix descriptor `descrA`.

```
integer(4) function cusparseSetMatFillMode(descrA, mode)
  type(cusparseMatDescr) :: descrA
  integer(4) :: mode
```

### 5.1.15. `cusparseGetMatFillMode`

This function returns the `FillMode` field of the matrix descriptor `descrA`.

```
integer(4) function cusparseGetMatFillMode(descrA)
  type(cusparseMatDescr) :: descrA
```

### 5.1.16. `cusparseSetMatDiagType`

This function sets the `DiagType` of the matrix descriptor `descrA`.

```
integer(4) function cusparseSetMatDiagType(descrA, type)
  type(cusparseMatDescr) :: descrA
  integer(4) :: type
```

### 5.1.17. `cusparseGetMatDiagType`

This function returns the `DiagType` of the matrix descriptor `descrA`.

```
integer(4) function cusparseGetMatDiagType(descrA)
  type(cusparseMatDescr) :: descrA
```

### 5.1.18. `cusparseSetMatIndexBase`

This function sets the `IndexBase` field of the matrix descriptor `descrA`.

```
integer(4) function cusparseSetMatIndexBase(descrA, base)
  type(cusparseMatDescr) :: descrA
  integer(4) :: base
```

### 5.1.19. `cusparseGetMatIndexBase`

This function returns the `IndexBase` field of the matrix descriptor `descrA`.

```
integer(4) function cusparseGetMatIndexBase(descrA)
  type(cusparseMatDescr) :: descrA
```

### 5.1.20. `cusparseCreateSolveAnalysisInfo`

This function creates and initializes the solve and analysis structure to default values. This function, and all functions which use the `cusparseSolveAnalysisInfo` type, are deprecated in CUDA 10.2 and removed from CUDA 11.0+.

```
integer(4) function cusparseCreateSolveAnalysisInfo(info)
  type(cusparseSolveAnalysisInfo) :: info
```

### 5.1.21. `cusparseDestroySolveAnalysisInfo`

This function destroys and releases any memory required by the structure. This function, and all functions which use the `cusparseSolveAnalysisInfo` type, are deprecated in CUDA 10.2 and removed from CUDA 11.0+.

```
integer(4) function cusparseDestroySolveAnalysisInfo(info)
  type(cusparseSolveAnalysisInfo) :: info
```

### 5.1.22. `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
```

### 5.1.23. `cusparseCreateHybMat`

This function creates and initializes the `hybA` opaque data structure. This function, and all functions which use the `cusparseHybMat` type, are deprecated in CUDA 10.2 and removed from CUDA 11.0+.

```
integer(4) function cusparseCreateHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

### 5.1.24. `cusparseDestroyHybMat`

This function destroys and releases any memory required by the `hybA` structure. This function, and all functions which use the `cusparseHybMat` type, are deprecated in CUDA 10.2 and removed from CUDA 11.0+.

```
integer(4) function cusparseDestroyHybMat(hybA)
  type(cusparseHybMat) :: hybA
```

### 5.1.25. `cusparseCreateCsrsv2Info`

This function creates and initializes the solve and analysis structure of `csrsv2` to default values.

```
integer(4) function cusparseCreateCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

### 5.1.26. `cusparseDestroyCsrsv2Info`

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrsv2Info(info)
  type(cusparseCsrsv2Info) :: info
```

### 5.1.27. `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
```

### 5.1.28. `cusparseDestroyCsric02Info`

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsric02Info(info)
  type(cusparseCsric02Info) :: info
```

### 5.1.29. `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
```

### 5.1.30. `cusparseDestroyCsrilu02Info`

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrilu02Info(info)
  type(cusparseCsrilu02Info) :: info
```

### 5.1.31. `cusparseCreateBsrsv2Info`

This function creates and initializes the solve and analysis structure of `bsrsv2` to default values.

```
integer(4) function cusparseCreateBsrsv2Info(info)
  type(cusparseBsrsv2Info) :: info
```

### 5.1.32. `cusparseDestroyBsrsv2Info`

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsv2Info(info)
```

```
type(cusparsv2Info) :: info
```

### 5.1.33. cusparseCreateBsrlic02Info

This function creates and initializes the solve and analysis structure of block incomplete Cholesky to default values.

```
integer(4) function cusparseCreateBsrlic02Info(info)
type(cusparseBsrlic02Info) :: info
```

### 5.1.34. cusparseDestroyBsrlic02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrlic02Info(info)
type(cusparseBsrlic02Info) :: info
```

### 5.1.35. 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
```

### 5.1.36. cusparseDestroyBsrilu02Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrilu02Info(info)
type(cusparseBsrilu02Info) :: info
```

### 5.1.37. cusparseCreateBsrsm2Info

This function creates and initializes the solve and analysis structure of bsrsm2 to default values.

```
integer(4) function cusparseCreateBsrsm2Info(info)
type(cusparseBsrsm2Info) :: info
```

### 5.1.38. cusparseDestroyBsrsm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyBsrsm2Info(info)
type(cusparseBsrsm2Info) :: info
```

### 5.1.39. cusparseCreateCsrgeemm2Info

This function creates and initializes the analysis structure of general sparse matrix-matrix multiplication.

```
integer(4) function cusparseCreateCsrgeemm2Info(info)
type(cusparseCsrgeemm2Info) :: info
```

### 5.1.40. cusparseDestroyCsrGemm2Info

This function destroys and releases any memory required by the structure.

```
integer(4) function cusparseDestroyCsrGemm2Info(info)
  type(cusparseCsrGemm2Info) :: info
```

### 5.1.41. cusparseCreateColorInfo

This function creates coloring information used in calls like CSRColor.

```
integer(4) function cusparseCreateColorInfo(info)
  type(cusparseColorInfo) :: info
```

### 5.1.42. cusparseDestroyColorInfo

This function destroys coloring information used in calls like CSRColor.

```
integer(4) function cusparseDestroyColorInfo(info)
  type(cusparseColorInfo) :: info
```

### 5.1.43. cusparseCreateCsrU2csrInfo

This function creates sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseCreateCsrU2csrInfo(info)
  type(cusparseCsrU2csrInfo) :: info
```

### 5.1.44. cusparseDestroyCsrU2csrInfo

This function destroys sorting information used in calls like CSRU2CSR.

```
integer(4) function cusparseDestroyCsrU2csrInfo(info)
  type(cusparseCsrU2csrInfo) :: info
```

## 5.2. CUSPARSE Level 1 Functions

This section contains interfaces for the level 1 sparse linear algebra functions that perform operations between dense and sparse vectors.

### 5.2.1. 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
```



## 5.2.2. 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
```

## 5.2.3. 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
```

## 5.2.4. 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
  complex(8), device :: alpha ! device or host variable
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(8), device :: y(*)
  integer :: idxBase
```

## 5.2.5. 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.  $res = \text{sum}(y * xVal(xInd))$

This function was deprecated in CUDA 10.1 and 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
```

### 5.2.6. 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 deprecated in CUDA 10.1 and 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
```

### 5.2.7. 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 deprecated in CUDA 10.1 and 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
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  complex(4), device :: y(*)
  complex(4), device :: res ! device or host variable
  integer :: idxBase
```

### 5.2.8. 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 deprecated in CUDA 10.1 and 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
```

### 5.2.9. 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.  $res = \sum(y * conj(xVal(xInd)))$

This function was deprecated in CUDA 10.1 and 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
```

### 5.2.10. `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 deprecated in CUDA 10.1 and 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
```

### 5.2.11. `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
```

### 5.2.12. `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
```

### 5.2.13. `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(cusparsHandle) :: handle
integer(4) :: nnz
complex(4), device :: y(*)
complex(4), device :: xVal(*)
integer(4), device :: xInd(*)
integer(4) :: idxBase

```

### 5.2.14. cusparsZgthr

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 cusparsZgthr(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparsHandle) :: handle
  integer(4) :: nnz
  complex(8), device :: y(*)
  complex(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase

```

### 5.2.15. cusparsSgthrz

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 cusparsSgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparsHandle) :: handle
  integer(4) :: nnz
  real(4), device :: y(*)
  real(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase

```

### 5.2.16. cusparsDgthrz

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 cusparsDgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparsHandle) :: handle
  integer(4) :: nnz
  real(8), device :: y(*)
  real(8), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase

```

### 5.2.17. cusparsCgthrz

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 cusparsCgthrz(handle, nnz, y, xVal, xInd, idxBase)
  type(cusparsHandle) :: handle
  integer(4) :: nnz
  complex(4), device :: y(*)
  complex(4), device :: xVal(*)
  integer(4), device :: xInd(*)
  integer(4) :: idxBase

```

## 5.2.18. 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
```

## 5.2.19. 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
```

## 5.2.20. 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
```

## 5.2.21. 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
```

## 5.2.22. 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
```

### 5.2.23. 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
```

### 5.2.24. 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(*)
  real(8), device :: y(*)
  real(8), device :: c, s ! device or host variable
  integer :: idxBase
```

## 5.3. 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.

### 5.3.1. 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(*)
```

### 5.3.2. 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  $(mb * blockDim) \times (nb * blockDim)$  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
  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(*)
```

### 5.3.3. 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  $(mb * blockDim) \times (nb * blockDim)$  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(*)
```

### 5.3.4. 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  $(mb * blockDim) \times (nb * blockDim)$  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(cusparsMatDescr) :: descr
complex(8), device :: bsrVal(*)
integer(4), device :: bsrRowPtr(*)
integer(4), device :: bsrColInd(*)
integer :: blockDim
complex(8), device :: x(*)
complex(8), device :: beta ! device or host variable
complex(8), device :: y(*)

```

### 5.3.5. `cusparsSbsrxmv`

**BSRXMLV** performs a BSRMV and a mask operation.

```

integer(4) function cusparsSbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparsHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparsMatDescr) :: 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(*)

```

### 5.3.6. `cusparsDbsrxmv`

**BSRXMLV** performs a BSRMV and a mask operation.

```

integer(4) function cusparsDbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparsHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparsMatDescr) :: 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(*)

```

### 5.3.7. `cusparsCbsrxmv`

**BSRXMLV** performs a BSRMV and a mask operation.

```

integer(4) function cusparsCbsrxmv(handle, dir, trans, sizeOfMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparsHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeOfMask
  integer :: mb, nb, nnzb

```



```

complex(4), device :: alpha ! device or host variable
type(cusparsMatDescr) :: 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(*)

```

### 5.3.8. `cusparsZbsrxmv`

BSRBMV performs a BSRMV and a mask operation.

```

integer(4) function cusparsZbsrxmv(handle, dir, trans, sizeofMask, mb, nb,
nnzb, alpha, descr, bsrVal, bsrMaskPtr, bsrRowPtr, bsrEndPtr, bsrColInd,
blockDim, x, beta, y)
  type(cusparsHandle) :: handle
  integer :: dir
  integer :: trans
  integer :: sizeofMask
  integer :: mb, nb, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparsMatDescr) :: 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(*)

```

### 5.3.9. `cusparsScsrmv`

CSRBMV 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  $m \times n$  sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with a call to `cusparsSpMV`

```

integer(4) function cusparsScsrmv(handle, trans, m, n, nnz, alpha, descr,
csrVal, csrRowPtr, csrColInd, x, beta, y)
  type(cusparsHandle) :: handle
  integer :: trans
  integer :: m, n, nnz
  real(4), device :: alpha ! device or host variable
  type(cusparsMatDescr) :: 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(*)

```

### 5.3.10. `cusparsDcsrmv`

CSRBMV 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

$m \times n$  sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with a call to `cusparseSpMV`

```
integer(4) function cusparseDcsrvm(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(*)
```

### 5.3.11. `cusparseCcsrvm`

CSRVM 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  $m \times n$  sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with a call to `cusparseSpMV`

```
integer(4) function cusparseCcsrvm(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(*)
```

### 5.3.12. `cusparseZcsrvm`

CSRVM 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  $m \times n$  sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with a call to `cusparseSpMV`

```
integer(4) function cusparseZcsrvm(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(*)
```

### 5.3.13. 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
```

### 5.3.14. 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
```

### 5.3.15. 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
```

### 5.3.16. 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
```

### 5.3.17. cusparsescrsv\_solve

This function performs the solve phase of crsv.

```
integer(4) function cusparsescrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  integer :: m
  real(4), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparsesolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

### 5.3.18. cusparsedcrsv\_solve

This function performs the solve phase of dcrsv.

```
integer(4) function cusparsedcrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  integer :: m
  real(8), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparsesolveAnalysisInfo) :: info
  real(8), device :: x(*)
  real(8), device :: y(*)
```

### 5.3.19. cusparsccrsv\_solve

This function performs the solve phase of crsv.

```
integer(4) function cusparsccrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  integer :: m
  complex(4), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*)
  integer(4), device :: csrColInd(*)
  type(cusparsesolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

### 5.3.20. cusparszcrsv\_solve

This function performs the solve phase of crsv.

```
integer(4) function cusparszcrsv_solve(handle, trans, m, alpha, descr,
csrVal, csrRowPtr, csrColInd, info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
```

```

integer :: m
complex(8), device :: alpha ! device or host variable
type(cusparsedMatDescr) :: descr
complex(8), device :: csrVal(*)
integer(4), device :: csrRowPtr(*)
integer(4), device :: csrColInd(*)
type(cusparsedSolveAnalysisInfo) :: info
complex(8), device :: x(*)
complex(8), device :: y(*)

```

### 5.3.21. cusparsedSgemvi\_bufferSize

This function returns the buffer size, in bytes, needed by cusparsedSgemvi.

```

integer(4) function cusparsedSgemvi_bufferSize(handle, transA, m, n, nnz,
pBufferSize)
  type(cusparsedHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize

```

### 5.3.22. cusparsedDgemvi\_bufferSize

This function returns the buffer size, in bytes, needed by cusparsedDgemvi.

```

integer(4) function cusparsedDgemvi_bufferSize(handle, transA, m, n, nnz,
pBufferSize)
  type(cusparsedHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize

```

### 5.3.23. cusparsedCgemvi\_bufferSize

This function returns the buffer size, in bytes, needed by cusparsedCgemvi.

```

integer(4) function cusparsedCgemvi_bufferSize(handle, transA, m, n, nnz,
pBufferSize)
  type(cusparsedHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize

```

### 5.3.24. cusparsedZgemvi\_bufferSize

This function returns the buffer size, in bytes, needed by cusparsedZgemvi.

```

integer(4) function cusparsedZgemvi_bufferSize(handle, transA, m, n, nnz,
pBufferSize)
  type(cusparsedHandle) :: handle
  integer :: transA
  integer :: m, n, nnz
  integer(4) :: pBufferSize

```

### 5.3.25. cusparsedSgemvi

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 cusparsedSgemvi(handle, transA, m, n, alpha, A, lda, nnz,
xVal, xInd, beta, y, idxBase, pBuffer)
  type(cusparsedHandle) :: 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

```

### 5.3.26. 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

```

### 5.3.27. 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

```

### 5.3.28. 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

```

### 5.3.29. `cusparseShybm`

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  $m \times n$  sparse matrix that is defined in the HYB storage format.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0.

```
integer(4) function cusparseShybm(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(*)
```

### 5.3.30. `cusparseDhybm`

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  $m \times n$  sparse matrix that is defined in the HYB storage format.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0.

```
integer(4) function cusparseDhybm(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(*)
```

### 5.3.31. `cusparseChybm`

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  $m \times n$  sparse matrix that is defined in the HYB storage format.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0.

```
integer(4) function cusparseChybm(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(*)
```

### 5.3.32. `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  $m \times n$  sparse matrix that is defined in the HYB storage format.

This function was deprecated in CUDA 10.1 and 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(*)
```

### 5.3.33. `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
```

### 5.3.34. `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
```

### 5.3.35. `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
```

### 5.3.36. `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
```

### 5.3.37. cusparseshybsv\_solve

This function performs the solve phase of `hybsv`.

```
integer(4) function cusparseshybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  real(4), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  type(cusparsesolveHybMat) :: hyb
  type(cusparsesolveAnalysisInfo) :: info
  real(4), device :: x(*)
  real(4), device :: y(*)
```

### 5.3.38. cusparsedhybsv\_solve

This function performs the solve phase of `hybsv`.

```
integer(4) function cusparsedhybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  real(8), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  type(cusparsesolveHybMat) :: hyb
  type(cusparsesolveAnalysisInfo) :: info
  real(8), device :: x(*)
  real(8), device :: y(*)
```

### 5.3.39. cusparseshybsv\_solve

This function performs the solve phase of `hybsv`.

```
integer(4) function cusparseshybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  complex(4), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  type(cusparsesolveHybMat) :: hyb
  type(cusparsesolveAnalysisInfo) :: info
  complex(4), device :: x(*)
  complex(4), device :: y(*)
```

### 5.3.40. cusparseshybsv\_solve

This function performs the solve phase of `hybsv`.

```
integer(4) function cusparseshybsv_solve(handle, trans, alpha, descr, hyb,
info, x, y)
  type(cusparsesolveHandle) :: handle
  integer :: trans
  complex(8), device :: alpha ! device or host variable
  type(cusparsesolveMatDescr) :: descr
  type(cusparsesolveHybMat) :: hyb
  type(cusparsesolveAnalysisInfo) :: info
  complex(8), device :: x(*)
  complex(8), device :: y(*)
```

### 5.3.41. `cusparseSbsrsv2_bufferSize`

This function returns the size of the buffer used in `bbsrsv2`.

```
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
```

### 5.3.42. `cusparseDbsrsv2_bufferSize`

This function returns the size of the buffer used in `bbsrsv2`.

```
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
  type(cusparseBsrsv2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted
```

### 5.3.43. `cusparseCbsrsv2_bufferSize`

This function returns the size of the buffer used in `bbsrsv2`.

```
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
```

### 5.3.44. `cusparseZbsrsv2_bufferSize`

This function returns the size of the buffer used in `bbsrsv2`.

```
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
```

### 5.3.45. 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
  type(cusparseBsrsv2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.3.46. 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(*)
```

### 5.3.47. 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(*)
```

### 5.3.48. 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
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
```

```
integer(4) :: blockDim
type(cusparsesBsrsv2Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)
```

### 5.3.49. 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(cusparsesHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparsesMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparsesBsrsv2Info) :: info
  real(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

### 5.3.50. cusparsesDbsrsv2\_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparsesDbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparsesHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparsesMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparsesBsrsv2Info) :: info
  real(8), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

### 5.3.51. cusparsesCbsrsv2\_solve

This function performs the solve phase of bsrsv2.

```
integer(4) function cusparsesCbsrsv2_solve(handle, dirA, transA, mb, nnzb,
alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, y, policy,
pBuffer)
  type(cusparsesHandle) :: handle
  integer :: dirA, transA, mb, nnzb
  complex(4), device :: alpha ! device or host variable
  type(cusparsesMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim
  type(cusparsesBsrsv2Info) :: info
  complex(4), device :: x(*), y(*)
  integer :: policy
  character, device :: pBuffer(*)
```

### 5.3.52. 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(*)
```

### 5.3.53. 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
```

### 5.3.54. 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
```

### 5.3.55. 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
```

### 5.3.56. `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
```

### 5.3.57. `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
```

### 5.3.58. `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(*)
```

### 5.3.59. `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(*)
```

### 5.3.60. 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(*)
```

### 5.3.61. 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(*)
```

### 5.3.62. 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(*)
```

### 5.3.63. 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(*)
```

### 5.3.64. 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(*)
```

### 5.3.65. 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(*)
```

### 5.3.66. 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
```

## 5.4. CUSPARSE Level 3 Functions

This section contains interfaces for the level 3 sparse linear algebra functions that perform operations between sparse and dense matrices.

### 5.4.1. cusparseScsrmm

CSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * B + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.2. `cusparseDcsrmm`

CSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * B + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
  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
```

### 5.4.3. `cusparseCcsrmm`

CSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * B + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.4. `cusparseZcsrmm`

CSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * B + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(A) = A$  or  $\text{op}(A) = A^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.5. `cusparseScsrmm2`

CSRMM2 performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.6. `cusparseDcsrmm2`

CSRMM2 performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.7. `cusparseCcsrmm2`

CSRMM2 performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
  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
```

### 5.4.8. `cusparseZcsrmm2`

CSRMM2 performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(A)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m \times k$  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 deprecated in CUDA 10.1 and 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
```

### 5.4.9. `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
```

### 5.4.10. `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(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparseSolveAnalysisInfo) :: info
```

### 5.4.11. 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
```

### 5.4.12. 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
```

### 5.4.13. 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
```

### 5.4.14. 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
```

### 5.4.15. 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
```

### 5.4.16. 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
```

### 5.4.17. 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
```

### 5.4.18. 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(cusparsCsrsm2Info) :: info
integer(8) :: pBufferSize

```

### 5.4.19. cusparsCcsrsm2\_bufferSizeExt

This function computes the work buffer size needed for the `cusparsCcsrsm2` routines.

```

integer(4) function cusparsCcsrsm2_bufferSizeExt(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(4) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(4), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(8) :: pBufferSize

```

### 5.4.20. cusparsZcsrsm2\_bufferSizeExt

This function computes the work buffer size needed for the `cusparsZcsrsm2` routines.

```

integer(4) function cusparsZcsrsm2_bufferSizeExt(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(8) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(8), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(8) :: pBufferSize

```

### 5.4.21. cusparsScsrsm2\_analysis

This function performs the analysis phase of `csrsm`.

```

integer(4) function cusparsScsrsm2_analysis(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(4) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(4), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type

```

### 5.4.22. cusparsDcsrsm2\_analysis

This function performs the analysis phase of `csrsm`.

```

integer(4) function cusparsDcsrsm2_analysis(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(8) :: alpha ! host or device variable

```

```

type(cusparsMatDescr) :: descrA
real(8), device :: csrValA(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
real(8), device :: B(ldb,*)
type(cusparsCsrsm2Info) :: info
integer(1), device :: pBuffer ! Any data type

```

### 5.4.23. cusparsCcsrsm2\_analysis

This function performs the analysis phase of csrsm.

```

integer(4) function cusparsCcsrsm2_analysis(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(4) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(4), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type

```

### 5.4.24. cusparsZcsrsm2\_analysis

This function performs the analysis phase of csrsm.

```

integer(4) function cusparsZcsrsm2_analysis(handle, algo, transA, transB,
m, nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  complex(8) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  complex(8), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type

```

### 5.4.25. cusparsScsrsm2\_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  $m \times m$  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 cusparsScsrsm2_solve(handle, algo, transA, transB, m,
nrhs, nnz, alpha, descrA, csrValA, csrRowPtrA, csrColIndA, B, ldb, info,
policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: algo, transA, transB, m, nrhs, nnz, ldb, policy
  real(4) :: alpha ! host or device variable
  type(cusparsMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  real(4), device :: B(ldb,*)
  type(cusparsCsrsm2Info) :: info
  integer(1), device :: pBuffer ! Any data type

```

### 5.4.26. 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  $m \times m$  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
```

### 5.4.27. 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  $m \times m$  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
```

### 5.4.28. 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  $m \times m$  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
```



### 5.4.29. `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
```

### 5.4.30. `cusparseSbsrmm`

BSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m_b \times k_b$  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
```

### 5.4.31. `cusparseDbsrmm`

BSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m_b \times k_b$  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
```

### 5.4.32. `cusparseCbsrmm`

BSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m_b \times k_b$  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(cusparsMatDescr) :: descrA
complex(4), device :: bsrValA(*), B(*), C(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
integer :: ldb, ldc

```

### 5.4.33. cusparsZbsrmm

BSRMM performs one of the matrix-matrix operations  $C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $\alpha$  and  $\beta$  are scalars.  $A$  is an  $m_b \times k_b$  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 cusparsZbsrmm(handle, dirA, transA, transB, mb, n, kb,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, B, ldb, beta,
C, ldc)
  type(cusparsHandle) :: handle
  integer :: dirA, transA, transB, mb, n, kb, nnzb, blockDim
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparsMatDescr) :: descrA
  complex(8), device :: bsrValA(*), B(*), C(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: ldb, ldc

```

### 5.4.34. cusparsSbsrsm2\_bufferSize

This function returns the size of the buffer used in `bsrsm2`.

```

integer(4) function cusparsSbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparsMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparsBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.4.35. cusparsDbsrsm2\_bufferSize

This function returns the size of the buffer used in `bsrsm2`.

```

integer(4) function cusparsDbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparsMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparsBsrsm2Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.4.36. cusparsCbsrsm2\_bufferSize

This function returns the size of the buffer used in `bsrsm2`.

```

integer(4) function cusparsCbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim

```

```

type(cusparsMatDescr) :: descrA
complex(4), device :: bsrValA(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
type(cusparsBsrsm2Info) :: info
integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.4.37. cusparsZbsrsm2\_bufferSize

This function returns the size of the buffer used in bsrsm2.

```

integer(4) function cusparsZbsrsm2_bufferSize(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info,
pBuffer)
type(cusparsHandle) :: handle
integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
type(cusparsMatDescr) :: descrA
complex(8), device :: bsrValA(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
type(cusparsBsrsm2Info) :: info
integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.4.38. cusparsSbsrsm2\_analysis

This function performs the analysis phase of bsrsm2.

```

integer(4) function cusparsSbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
type(cusparsHandle) :: handle
integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
type(cusparsMatDescr) :: descrA
real(4), device :: bsrValA(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
type(cusparsBsrsm2Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)

```

### 5.4.39. cusparsDbsrsm2\_analysis

This function performs the analysis phase of bsrsm2.

```

integer(4) function cusparsDbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
type(cusparsHandle) :: handle
integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
type(cusparsMatDescr) :: descrA
real(8), device :: bsrValA(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
type(cusparsBsrsm2Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)

```

### 5.4.40. cusparsCbsrsm2\_analysis

This function performs the analysis phase of bsrsm2.

```

integer(4) function cusparsCbsrsm2_analysis(handle, dirA, transA, transX,
mb, n, nnzb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy,
pBuffer)
type(cusparsHandle) :: handle
integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
type(cusparsMatDescr) :: descrA
complex(4), device :: bsrValA(*)

```

```
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
type(cusparsesBsrsm2Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)
```

### 5.4.41. 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(cusparsesHandle) :: handle
  integer(4) :: dirA, transA, transX, mb, n, nnzb, blockDim
  type(cusparsesMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  type(cusparsesBsrsm2Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.4.42. 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(cusparsesHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(4), device :: alpha ! device or host variable
  type(cusparsesMatDescr) :: descrA
  real(4), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparsesBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

### 5.4.43. 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(cusparsesHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  real(8), device :: alpha ! device or host variable
  type(cusparsesMatDescr) :: descrA
  real(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparsesBsrsm2Info) :: info
  character, device :: pBuffer(*)
```

### 5.4.44. 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(cusparsesHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
```

```

complex(4), device :: alpha ! device or host variable
type(cusparsMatDescr) :: descrA
complex(4), device :: bsrValA(*), x(*), y(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
integer :: blockDim, policy, ldx, ldy
type(cusparsBsrsm2Info) :: info
character, device :: pBuffer(*)

```

### 5.4.45. `cusparsZbsrsm2_solve`

This function performs the solve phase of `bsrsm2`.

```

integer(4) function cusparsZbsrsm2_solve(handle, dirA, transA, transX, mb, n,
nnzb, alpha, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, x, ldx,
y, ldy, policy, pBuffer)
  type(cusparsHandle) :: handle
  integer :: dirA, transA, transX, mb, n, nnzb
  complex(8), device :: alpha ! device or host variable
  type(cusparsMatDescr) :: descrA
  complex(8), device :: bsrValA(*), x(*), y(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer :: blockDim, policy, ldx, ldy
  type(cusparsBsrsm2Info) :: info
  character, device :: pBuffer(*)

```

### 5.4.46. `cusparsXbsrsm2_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 cusparsXbsrsm2_zeroPivot(handle, info, position)
  type(cusparsHandle) :: handle
  type(cusparsBsrsm2Info) :: info
  integer(4), device :: position ! device or host variable

```

### 5.4.47. `cusparsSgemmi`

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 `cusparsSpMM` instead.

```

integer(4) function cusparsSgemmi(handle, m, n, k, nnz, alpha, A, lda,
cscValB, cscColPtrB, cscRowIndB, beta, C, ldc)
  type(cusparsHandle) :: 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(*)

```

### 5.4.48. `cusparsDgemmi`

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(*)
  real(8), device :: C(ldc,*)
  integer(4), device :: cscColPtrB(*), cscRowIndB(*)
```

### 5.4.49. `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(*)
```

### 5.4.50. `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(*)
```

## 5.5. CUSPARSE Extra Functions

This section contains interfaces for the extra functions that are used to manipulate sparse matrices.

### 5.5.1. `cusparseXcsrgeamNnz`

`cusparseXcsrgeamNnz` computes the number of nonzero elements which will be produced by CSRGEAM.

This function was deprecated in CUDA 10.1 and 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
```

### 5.5.2. `cusparseScsrgeam`

CSRGEAM performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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 deprecated in CUDA 10.1 and 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(*)
```

### 5.5.3. `cusparseDcsrgeam`

CSRGEAM performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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 deprecated in CUDA 10.1 and 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(*)
```

### 5.5.4. `cusparseCcsrgeam`

CSRGEAM performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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 deprecated in CUDA 10.1 and 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(*)
```

### 5.5.5. `cusparseZcsrgeam`

CSRGEAM performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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 deprecated in CUDA 10.1 and 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(*)
```



### 5.5.6. `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  $m \times n$  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
```

### 5.5.7. `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  $m \times n$  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(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes
```

### 5.5.8. `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  $m \times n$  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(cusparsMatDescr):: descrA, descrB, descrC
complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
integer(8) :: pBufferSizeInBytes

```

### 5.5.9. `cusparsZcsrgeam2_bufferSizeExt`

This function determines the work buffer size for `cusparsZcsrgeam2`. CSRGEAM2 performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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}`. `cusparsXcsrgeam2Nnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

```

integer(4) function cusparsZcsrgeam2_bufferSizeExt(handle, m, n, alpha,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, beta, descrB, nnzB,
csrValB, csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC,
pBufferSizeInBytes)
  type(cusparsHandle) :: handle
  integer :: m, n, nnzA, nnzB
  complex(8), device :: alpha, beta ! device or host variable
  type(cusparsMatDescr):: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)
  integer(8) :: pBufferSizeInBytes

```

### 5.5.10. `cusparsXcsrgeam2Nnz`

`cusparsXcsrgeam2Nnz` computes the number of nonzero elements which will be produced by CSRGEAM2.

```

integer(4) function cusparsXcsrgeam2Nnz(handle, m, n, descrA, nnzA,
csrRowPtrA, csrColIndA, descrB, nnzB, csrRowPtrB, csrColIndB, descrC,
csrRowPtrC, nnzTotalDevHostPtr, pBuffer)
  type(cusparsHandle) :: handle
  type(cusparsMatDescr) :: 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(*)

```

### 5.5.11. `cusparsCcsrgeam2`

CSRGEAM2 performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ ,  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times n$  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}`. `cusparsXcsrgeam2Nnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

```

integer(4) function cusparsCcsrgeam2(handle, m, n, alpha, descrA, nnzA,
csrValA, csrRowPtrA, csrColIndA, beta, descrB, nnzB, csrValB, csrRowPtrB,
csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC, pBuffer)
  type(cusparsHandle) :: handle
  integer :: m, n, nnzA, nnzB
  real(4), device :: alpha, beta ! device or host variable
  type(cusparsMatDescr):: 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

```

### 5.5.12. cusparseDcsrgeam2

CSRGEAM2 performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ , alpha and beta are scalars. A, B, and C are  $m \times n$  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(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

```

### 5.5.13. cusparseCcsrgeam2

CSRGEAM2 performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ , alpha and beta are scalars. A, B, and C are  $m \times n$  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

```

### 5.5.14. cusparseZcsrgeam2

CSRGEAM2 performs the matrix-matrix operation  $C := \alpha * A + \beta * B$ , alpha and beta are scalars. A, B, and C are  $m \times n$  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

```

### 5.5.15. cusparseXcsrgermmNnz

`cusparseXcsrgermmNnz` computes the number of nonzero elements which will be produced by CSR GEMM.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with the `cusparseXcsrgermm2Nnz` routines

```

integer(4) function cusparseXcsrgermmNnz(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

```

### 5.5.16. cusparseScsrgemm

CSR GEMM performs the matrix-matrix operation  $C := \text{op}(A) * \text{op}(B)$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparseXcsrgermmNnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

This function was deprecated in CUDA 10.1 and 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(*)

```

### 5.5.17. cusparseDcsrgemm

CSR GEMM performs the matrix-matrix operation  $C := \text{op}(A) * \text{op}(B)$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparseXcsrgermmNnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

This function was deprecated in CUDA 10.1 and 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(cusparsMatDescr) :: descrA, descrB, descrC
real(8), device :: csrValA(*), csrValB(*), csrValC(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)

```

### 5.5.18. `cusparsCcsrgemm`

CSRGEEMM performs the matrix-matrix operation  $C := \text{op}(A) * \text{op}(B)$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparsXcsrgemmNnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with the `cusparsCcsrgemm2` routines

```

integer(4) function cusparsCcsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparsHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparsMatDescr) :: descrA, descrB, descrC
  complex(4), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)

```

### 5.5.19. `cusparsZcsrgemm`

CSRGEEMM performs the matrix-matrix operation  $C := \text{op}(A) * \text{op}(B)$ , where  $\text{op}(X)$  is one of  $\text{op}(X) = X$  or  $\text{op}(X) = X^{**T}$ ,  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparsXcsrgemmNnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

This function was deprecated in CUDA 10.1 and removed in CUDA 11.0. It should be replaced with the `cusparsZcsrgemm2` routines

```

integer(4) function cusparsZcsrgemm(handle, transA, transB, m, n, k,
descrA, nnzA, csrValA, csrRowPtrA, csrColIndA, descrB, nnzB, csrValB,
csrRowPtrB, csrColIndB, descrC, csrValC, csrRowPtrC, csrColIndC)
  type(cusparsHandle) :: handle
  integer(4) :: transA, transB, m, n, k, nnzA, nnzB
  type(cusparsMatDescr) :: descrA, descrB, descrC
  complex(8), device :: csrValA(*), csrValB(*), csrValC(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrC(*), csrColIndC(*)

```

### 5.5.20. `cusparsScsrgemm2_bufferSizeExt`

This function returns the size of the buffer used in `csrgemm2`.

```

integer(4) function cusparsScsrgemm2_bufferSizeExt(handle, m, n, k,
alpha, descrA, nnzA, csrRowPtrA, csrColIndA, descrB,
nnzB, csrRowPtrB, csrColIndB, beta, descrD, nnzD, csrRowPtrD,
csrColIndD, info, pBufferSizeInBytes)
  type(cusparsHandle) :: handle
  real(4), device :: alpha, beta ! device or host variable
  integer :: m, n, k, nnzA, nnzB, nnzD
  type(cusparsMatDescr) :: descrA, descrB, descrD
  integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
csrColIndB(*), csrRowPtrD(*), csrColIndD(*)

```

```

type(cusparsedcsrgermm2Info) :: info
integer(8) :: pBufferSizeInBytes

```

### 5.5.21. cusparsedcsrgermm2\_bufferSizeExt

This function returns the size of the buffer used in `csrgermm2`.

```

integer(4) function cusparsedcsrgermm2_bufferSizeExt(handle, m, n, k,
    alpha, descrA, nnzA, csrRowPtrA, csrColIndA, descrB,
    nnzB, csrRowPtrB, csrColIndB, beta, descrD, nnzD, csrRowPtrD,
    csrColIndD, info, pBufferSizeInBytes)
    type(cusparsedHandle) :: handle
    real(8), device :: alpha, beta ! device or host variable
    integer :: m, n, k, nnzA, nnzB, nnzD
    type(cusparsedMatDescr) :: descrA, descrB, descrD
    integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
    csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
    type(cusparsedCsrgermm2Info) :: info
    integer(8) :: pBufferSizeInBytes

```

### 5.5.22. cusparsedccsrgermm2\_bufferSizeExt

This function returns the size of the buffer used in `csrgermm2`.

```

integer(4) function cusparsedccsrgermm2_bufferSizeExt(handle, m, n, k,
    alpha, descrA, nnzA, csrRowPtrA, csrColIndA, descrB,
    nnzB, csrRowPtrB, csrColIndB, beta, descrD, nnzD, csrRowPtrD,
    csrColIndD, info, pBufferSizeInBytes)
    type(cusparsedHandle) :: handle
    complex(4), device :: alpha, beta ! device or host variable
    integer :: m, n, k, nnzA, nnzB, nnzD
    type(cusparsedMatDescr) :: descrA, descrB, descrD
    integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
    csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
    type(cusparsedCsrgermm2Info) :: info
    integer(8) :: pBufferSizeInBytes

```

### 5.5.23. cusparsedzcsrgermm2\_bufferSizeExt

This function returns the size of the buffer used in `csrgermm2`.

```

integer(4) function cusparsedzcsrgermm2_bufferSizeExt(handle, m, n, k,
    alpha, descrA, nnzA, csrRowPtrA, csrColIndA, descrB,
    nnzB, csrRowPtrB, csrColIndB, beta, descrD, nnzD, csrRowPtrD,
    csrColIndD, info, pBufferSizeInBytes)
    type(cusparsedHandle) :: handle
    complex(8), device :: alpha, beta ! device or host variable
    integer :: m, n, k, nnzA, nnzB, nnzD
    type(cusparsedMatDescr) :: descrA, descrB, descrD
    integer(4), device :: csrRowPtrA(*), csrColIndA(*), csrRowPtrB(*),
    csrColIndB(*), csrRowPtrD(*), csrColIndD(*)
    type(cusparsedCsrgermm2Info) :: info
    integer(8) :: pBufferSizeInBytes

```

### 5.5.24. cusparsedxcsrgermm2Nnz

`cusparsedxcsrgermm2Nnz` computes the number of nonzero elements which will be produced by `CSRGERMM2`.

```

integer(4) function cusparsedxcsrgermm2Nnz(handle, m, n, k,
    descrA, nnzA, csrRowPtrA, csrColIndA, descrB, nnzB, csrRowPtrB,
    csrColIndB, descrD, nnzD, csrRowPtrD, csrColIndD, descrC,
    csrRowPtrC, nnzTotalDevHostPtr, info, pBuffer)
    type(cusparsedHandle) :: handle

```

```

type(cusparsedMatDescr) :: descrA, descrB, descrD, descrC
type(cusparsedCsrGemm2Info) :: 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(*)

```

### 5.5.25. cusparseScsrGemm2

CSR GEMM2 performs the matrix-matrix operation  $C := \alpha * A * B + \beta * D$  where  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparsedXcsrGemm2Nnz` 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(cusparsedHandle) :: handle
type(cusparsedMatDescr) :: descrA, descrB, descrD, descrC
type(cusparsedCsrGemm2Info) :: 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(*)

```

### 5.5.26. cusparseDcsrGemm2

CSR GEMM2 performs the matrix-matrix operation  $C := \alpha * A * B + \beta * D$  where  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparsedXcsrGemm2Nnz` 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(cusparsedHandle) :: handle
type(cusparsedMatDescr) :: descrA, descrB, descrD, descrC
type(cusparsedCsrGemm2Info) :: 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(*)

```

### 5.5.27. cusparseCcsrGemm2

CSR GEMM2 performs the matrix-matrix operation  $C := \alpha * A * B + \beta * D$  where  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparseXcsrgermm2Nnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

```
integer(4) function cusparseCcsrgermm2(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(cusparseCsrgermm2Info) :: info
  integer :: m, n, k, nnzA, nnzB, nnzD
  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(*)
```

## 5.5.28. cusparseZcsrgermm2

CSRGERMM2 performs the matrix-matrix operation  $C := \alpha * A * B + \beta * D$  where  $\alpha$  and  $\beta$  are scalars.  $A$ ,  $B$ , and  $C$  are  $m \times k$ ,  $k \times n$ , and  $m \times n$  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}`. `cusparseXcsrgermm2Nnz` should be used to determine `csrRowPtrC` and the number of nonzero elements in the result.

```
integer(4) function cusparseZcsrgermm2(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(cusparseCsrgermm2Info) :: 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(*)
```

## 5.6. CUSPARSE Preconditioning Functions

This section contains interfaces for the preconditioning functions that are used in processing sparse matrices.

### 5.6.1. cusparseScsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting.  $A$  is an  $m \times n$  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(cusparsedMatDescr) :: descrA
real(4), device :: csrValM(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
type(cusparsedSolveAnalysisInfo) :: info
```

### 5.6.2. cusparsedDcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an  $m \times n$  Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`.

```
integer(4) function cusparsedDcsric0(handle, trans, m, descrA,
csrValM, csrRowPtrA, csrColIndA, info)
type(cusparsedHandle) :: handle
integer(4) :: trans, m
type(cusparsedMatDescr) :: descrA
real(8), device :: csrValM(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
type(cusparsedSolveAnalysisInfo) :: info
```

### 5.6.3. cusparsedCcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an  $m \times n$  Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`.

```
integer(4) function cusparsedCcsric0(handle, trans, m, descrA,
csrValM, csrRowPtrA, csrColIndA, info)
type(cusparsedHandle) :: handle
integer(4) :: trans, m
type(cusparsedMatDescr) :: descrA
complex(4), device :: csrValM(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
type(cusparsedSolveAnalysisInfo) :: info
```

### 5.6.4. cusparsedZcsric0

CSRIC0 computes the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an  $m \times n$  Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`.

```
integer(4) function cusparsedZcsric0(handle, trans, m, descrA,
csrValM, csrRowPtrA, csrColIndA, info)
type(cusparsedHandle) :: handle
integer(4) :: trans, m
type(cusparsedMatDescr) :: descrA
complex(8), device :: csrValM(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
type(cusparsedSolveAnalysisInfo) :: info
```

### 5.6.5. cusparseScsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times n$  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
```

### 5.6.6. cusparseDcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times n$  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
```

### 5.6.7. cusparseCcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times n$  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
```

### 5.6.8. cusparseZcsrilu0

CSRILU0 computes the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times n$  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
```

### 5.6.9. `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 deprecated in CUDA 10.1 and 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(*)
```

### 5.6.10. `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 deprecated in CUDA 10.1 and 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(*)
```

### 5.6.11. `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 deprecated in CUDA 10.1 and 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(*)
```

### 5.6.12. `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 deprecated in CUDA 10.1 and 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(*)
```

### 5.6.13. `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
```

### 5.6.14. `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
```

### 5.6.15. `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
```

### 5.6.16. `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
```

### 5.6.17. `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(*)
```

### 5.6.18. 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(*)
```

### 5.6.19. 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(*)
```

### 5.6.20. 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(*)
```

### 5.6.21. 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
```

### 5.6.22. 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
```

### 5.6.23. 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
```

### 5.6.24. 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
```

### 5.6.25. 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(*)

```

### 5.6.26. 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(*)

```

### 5.6.27. 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(*)

```

### 5.6.28. 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(*)

```

### 5.6.29. `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
```

### 5.6.30. `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
```

### 5.6.31. `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
  complex(4), device :: dl(*), d(*), du(*), x(*)
  integer(8) :: pBufferSizeInBytes
```

### 5.6.32. `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
```

### 5.6.33. `cusparseSgtsv2StridedBatch`

`Sgtsv2StridedBatch` 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, batchSize
real(4), device :: dl(*), d(*), du(*), x(*)
character(1), device :: pBuffer(*)
```

### 5.6.34. cusparseDgtsv2StridedBatch

Dgtsv2StridedBatch 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, batchSize, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchSize
  real(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

### 5.6.35. 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, batchSize, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchSize
  complex(4), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

### 5.6.36. 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, batchSize, pBuffer)
  type(cusparseHandle) :: handle
  integer(4) :: m, batchCount, batchSize
  complex(8), device :: dl(*), d(*), du(*), x(*)
  character(1), device :: pBuffer(*)
```

### 5.6.37. cusparseSgtsvInterleavedBatch\_buffersize

SgtsvInterleavedBatch\_buffersize returns the size of the buffer, in bytes, required in SgtsvInterleavedBatch().

```
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
```

### 5.6.38. 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
```

### 5.6.39. 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
```

### 5.6.40. 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
```

### 5.6.41. 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(*)
```

### 5.6.42. 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(*)
```

### 5.6.43. 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(*)
```

### 5.6.44. 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(*)
```

### 5.6.45. `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
```

### 5.6.46. `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
```

### 5.6.47. `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
```

### 5.6.48. `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
```

### 5.6.49. `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(*)
```

## 5.6.50. 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(*)
```

## 5.6.51. 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(*)
```

## 5.6.52. 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(*)
```

### 5.6.53. 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
```

### 5.6.54. 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
```

### 5.6.55. 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
```

### 5.6.56. 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
```

### 5.6.57. 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(*)
```

### 5.6.58. 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(*)
```

### 5.6.59. 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(*)
```

### 5.6.60. 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(*)
```

### 5.6.61. `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(cusparseCsr02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.62. `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(cusparseCsr02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.63. `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(cusparseCsr02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.64. `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(cusparseCsr02Info) :: info
```



```
integer(4) :: policy
character(c_char), device :: pBuffer(*)
```

### 5.6.65. 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(cusparseCsrlic02Info) :: info
  integer(4), device :: position ! device or host variable
```

### 5.6.66. 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
```

### 5.6.67. 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
```

### 5.6.68. 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
```

### 5.6.69. 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(cusparsCsrilu02Info) :: info
integer :: enable_boost
real(8), device :: tol ! device or host variable
complex(8), device :: boost_val ! device or host variable

```

## 5.6.70. cusparsScsrilu02\_bufferSize

This function returns the size of the buffer used in csrlu02.

```

integer(4) function cusparsScsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: m, nnz
  type(cusparsMatDescr) :: descrA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparsCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

## 5.6.71. cusparsDcsrilu02\_bufferSize

This function returns the size of the buffer used in csrlu02.

```

integer(4) function cusparsDcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: m, nnz
  type(cusparsMatDescr) :: descrA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparsCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

## 5.6.72. cusparsCcsrilu02\_bufferSize

This function returns the size of the buffer used in csrlu02.

```

integer(4) function cusparsCcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: m, nnz
  type(cusparsMatDescr) :: descrA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparsCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

## 5.6.73. cusparsZcsrilu02\_bufferSize

This function returns the size of the buffer used in csrlu02.

```

integer(4) function cusparsZcsrilu02_bufferSize(handle, m, nnz, descrA,
csrValA, csrRowPtrA, csrColIndA, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: m, nnz
  type(cusparsMatDescr) :: descrA
  complex(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)
  type(cusparsCsrilu02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.6.74. 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(*)
```

### 5.6.75. 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(*)
```

### 5.6.76. 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(*)
```

### 5.6.77. 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(*)
```

### 5.6.78. cusparseScsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times m$  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(*)
```

### 5.6.79. cusparseDcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times m$  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(*)
```

### 5.6.80. cusparseCcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times m$  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(*)
```

### 5.6.81. cusparseZcsrilu02

CSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $m \times m$  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(cusparsMatDescr) :: descrA
complex(8), device :: csrValA(*)
integer(4), device :: csrRowPtrA(*), csrColIndA(*)
type(cusparsCsrilu02Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)

```

### 5.6.82. cusparsXcsrilu02\_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 cusparsXcsrilu02_zeroPivot(handle, info, position)
  type(cusparsHandle) :: handle
  type(cusparsCsrilu02Info) :: info
  integer(4), device :: position ! device or host variable

```

### 5.6.83. cusparsSbsric02\_bufferSize

This function returns the size of the buffer used in bsric02.

```

integer(4) function cusparsSbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
  bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsBsrlic02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.6.84. cusparsDbsric02\_bufferSize

This function returns the size of the buffer used in bsric02.

```

integer(4) function cusparsDbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
  bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsBsrlic02Info) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

### 5.6.85. cusparsCbsric02\_bufferSize

This function returns the size of the buffer used in bsric02.

```

integer(4) function cusparsCbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
  bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim

```

```

type(cusparsesBsrinfo) :: info
integer(4) :: pBufferSize ! integer(8) also accepted

```

## 5.6.86. cusparsesZbsric02\_bufferSize

This function returns the size of the buffer used in bsric02.

```

integer(4) function cusparsesZbsric02_bufferSize(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, pBufferSize)
  type(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrinfo) :: info
  integer(4) :: pBufferSize ! integer(8) also accepted

```

## 5.6.87. cusparsesSbsric02\_analysis

This function performs the analysis phase of bsric02.

```

integer(4) function cusparsesSbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrinfo) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)

```

## 5.6.88. cusparsesDbsric02\_analysis

This function performs the analysis phase of bsric02.

```

integer(4) function cusparsesDbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrinfo) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)

```

## 5.6.89. cusparsesCbsric02\_analysis

This function performs the analysis phase of bsric02.

```

integer(4) function cusparsesCbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  complex(4), device :: bsrValA(*)

```

```
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
integer(4) :: blockDim
type(cusparsesBsrC02Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)
```

## 5.6.90. cusparseZbsric02\_analysis

This function performs the analysis phase of bsrC02.

```
integer(4) function cusparseZbsric02_analysis(handle, dirA, mb, nnzb, descrA,
bsrValA, bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrC02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

## 5.6.91. 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(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  real(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrC02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

## 5.6.92. 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(cusparsesHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsesMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsesBsrC02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.93. cusparseCbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an  $(mb \times blockDim) \times (mb \times blockDim)$  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(cusparseBsrlic02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.94. cusparseZbsric02

BSRIC02 performs the solve phase of the incomplete-Cholesky factorization with zero fill-in and no pivoting. A is an  $(mb \times blockDim) \times (mb \times blockDim)$  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(cusparseBsrlic02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)
```

### 5.6.95. 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(cusparseBsrlic02Info) :: info
  integer(4), device :: position ! device or host variable
```

### 5.6.96. 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
```

### 5.6.97. 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
```

### 5.6.98. 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
```

### 5.6.99. 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
```

### 5.6.100. 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
```

### 5.6.101. 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
```

### 5.6.102. 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
```

### 5.6.103. 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
```

### 5.6.104. 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(*)
```

### 5.6.105. 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(*)
```

### 5.6.106. 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(*)
```

### 5.6.107. 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(*)
```

### 5.6.108. 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(cusparsMatDescr) :: descrA
real(4), device :: bsrValA(*)
integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
integer(4) :: blockDim
type(cusparsBsrilu02Info) :: info
integer(4) :: policy
character(c_char), device :: pBuffer(*)

```

### 5.6.109. cusparsDbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $(mb \times blockDim) \times (mb \times blockDim)$  sparse matrix that is defined in BSR storage format by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA`.

```

integer(4) function cusparsDbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  real(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)

```

### 5.6.110. cusparsCbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $(mb \times blockDim) \times (mb \times blockDim)$  sparse matrix that is defined in BSR storage format by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA`.

```

integer(4) function cusparsCbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  complex(4), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsBsrilu02Info) :: info
  integer(4) :: policy
  character(c_char), device :: pBuffer(*)

```

### 5.6.111. cusparsZbsrilu02

BSRILU02 performs the solve phase of the incomplete-LU factorization with zero fill-in and no pivoting. A is an  $(mb \times blockDim) \times (mb \times blockDim)$  sparse matrix that is defined in BSR storage format by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA`.

```

integer(4) function cusparsZbsrilu02(handle, dirA, mb, nnzb, descrA, bsrValA,
bsrRowPtrA, bsrColIndA, blockDim, info, policy, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: dirA
  integer(4) :: mb, nnzb
  type(cusparsMatDescr) :: descrA
  complex(8), device :: bsrValA(*)
  integer(4), device :: bsrRowPtrA(*), bsrColIndA(*)
  integer(4) :: blockDim
  type(cusparsBsrilu02Info) :: info

```

```
integer(4) :: policy
character(c_char), device :: pBuffer(*)
```

### 5.6.112. 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
```

## 5.7. CUSPARSE Reordering Functions

This section contains interfaces for the reordering functions that are used to manipulate sparse matrices.

### 5.7.1. 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
```

### 5.7.2. 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
```

### 5.7.3. 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
```

### 5.7.4. 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
```

## 5.8. 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.

### 5.8.1. 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(*)
```

## 5.8.2. 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(*)
```

## 5.8.3. 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(*)
```

## 5.8.4. 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(*)
```

## 5.8.5. 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(*)
```

### 5.8.6. `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
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

### 5.8.7. `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(*)
```

### 5.8.8. `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(*)
```

### 5.8.9. `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(*)
```



## 5.8.10. 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
```

## 5.8.11. 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
```

## 5.8.12. 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
```

## 5.8.13. 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
```

### 5.8.14. `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
```

### 5.8.15. `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(*)
```

### 5.8.16. `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(*)
```

### 5.8.17. `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(*)
```

### 5.8.18. `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(*)
```

### 5.8.19. `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(*)
```

### 5.8.20. `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 deprecated in CUDA 10.1 and 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(*)
```

### 5.8.21. `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 deprecated in CUDA 10.1 and 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(*)
```

## 5.8.22. `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 deprecated in CUDA 10.1 and 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(*)
```

## 5.8.23. `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 deprecated in CUDA 10.1 and 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(*)
```

## 5.8.24. `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
```

## 5.8.25. `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
```

### 5.8.26. 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, cscValA,
cscRowPtrA, cscColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowPtrA(*), cscColIndA(*)
```

### 5.8.27. 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, cscValA,
cscRowPtrA, cscColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  real(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowPtrA(*), cscColIndA(*)
```

### 5.8.28. 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, cscValA,
cscRowPtrA, cscColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(4), device :: cscValA(*), A(*)
  integer(4), device :: cscRowPtrA(*), cscColIndA(*)
```

### 5.8.29. 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, cscValA,
cscRowPtrA, cscColIndA, A, lda)
  type(cusparseHandle) :: handle
  integer(4) :: m, n, lda
  type(cusparseMatDescr) :: descrA
  complex(8), device :: cscValA(*), A(*)
  integer(4), device :: cscRowPtrA(*), cscColIndA(*)
```

### 5.8.30. `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(*)
```

### 5.8.31. `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(*)
```

### 5.8.32. `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(*)
```

### 5.8.33. `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(*)
```

### 5.8.34. `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(*)
```

### 5.8.35. `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(*)
```

### 5.8.36. `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(*)
```

### 5.8.37. `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(*)
```

### 5.8.38. `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(*)
```

### 5.8.39. `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(*)
```

### 5.8.40. `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(*)
```

### 5.8.41. `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(*)
```

### 5.8.42. `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(*)
```

### 5.8.43. `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(*)
```



### 5.8.44. `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(*)
```

### 5.8.45. `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(*)
```

### 5.8.46. `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
  real(4), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)
```

### 5.8.47. `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(*)
```

### 5.8.48. `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(cusparsHybMat) :: hybA
complex(4), device :: cscValA(*)
integer(4), device :: cscRowIndA(*), cscColPtrA(*)

```

### 5.8.49. `cusparsZhyb2csc`

This function converts the sparse matrix A in HYB format into a sparse matrix in CSC format.

```

integer(4) function cusparsZhyb2csc(handle, descrA, hybA, cscValA, cscRowIndA,
cscColPtrA)
  type(cusparsHandle) :: handle
  type(cusparsMatDescr) :: descrA
  type(cusparsHybMat) :: hybA
  complex(8), device :: cscValA(*)
  integer(4), device :: cscRowIndA(*), cscColPtrA(*)

```

### 5.8.50. `cusparsShyb2csr`

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```

integer(4) function cusparsShyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparsHandle) :: handle
  type(cusparsMatDescr) :: descrA
  type(cusparsHybMat) :: hybA
  real(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)

```

### 5.8.51. `cusparsDhyb2csr`

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```

integer(4) function cusparsDhyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparsHandle) :: handle
  type(cusparsMatDescr) :: descrA
  type(cusparsHybMat) :: hybA
  real(8), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)

```

### 5.8.52. `cusparsChyb2csr`

This function converts the sparse matrix A in HYB format into a sparse matrix in CSR format.

```

integer(4) function cusparsChyb2csr(handle, descrA, hybA, csrValA, csrRowPtrA,
csrColIndA)
  type(cusparsHandle) :: handle
  type(cusparsMatDescr) :: descrA
  type(cusparsHybMat) :: hybA
  complex(4), device :: csrValA(*)
  integer(4), device :: csrRowPtrA(*), csrColIndA(*)

```

### 5.8.53. `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(*)
```

### 5.8.54. `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
```

### 5.8.55. `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
```

### 5.8.56. `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
```

### 5.8.57. `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(cusparsHandle) :: handle
type(cusparsMatDescr) :: descrA
type(cusparsHybMat) :: hybA
complex(8), device :: A(*)
integer(4) :: lda

```

### 5.8.58. cusparsSnnz

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 cusparsSnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparsHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparsMatDescr) :: descrA
  real(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable

```

### 5.8.59. cusparsDnnz

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 cusparsDnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparsHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparsMatDescr) :: descrA
  real(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable

```

### 5.8.60. cusparsCnnz

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 cusparsCnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparsHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparsMatDescr) :: descrA
  complex(4), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable

```

### 5.8.61. cusparsZnnz

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 cusparsZnnz(handle, dirA, m, n, descrA, A, lda,
nnzPerRowColumn, nnzTotalDevHostPtr)
  type(cusparsHandle) :: handle
  integer :: dirA, m, n, lda
  type(cusparsMatDescr) :: descrA
  complex(8), device :: A(*)
  integer(4), device :: nnzPerRowColumn(*)
  integer(4), device :: nnzTotalDevHostPtr ! device or host variable

```

### 5.8.62. `cusparseSgebsr2gebbsc_bufferSize`

This function returns the size of the buffer used in `gebsr2gebbsc`.

```
integer(4) function cusparseSgebsr2gebbsc_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
```

### 5.8.63. `cusparseDgebsr2gebbsc_bufferSize`

This function returns the size of the buffer used in `gebsr2gebbsc`.

```
integer(4) function cusparseDgebsr2gebbsc_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
```

### 5.8.64. `cusparseCgebsr2gebbsc_bufferSize`

This function returns the size of the buffer used in `gebsr2gebbsc`.

```
integer(4) function cusparseCgebsr2gebbsc_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
```

### 5.8.65. `cusparseZgebsr2gebbsc_bufferSize`

This function returns the size of the buffer used in `gebsr2gebbsc`.

```
integer(4) function cusparseZgebsr2gebbsc_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
```

### 5.8.66. `cusparseSgebsr2gebbsc`

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 cusparseSgebsr2gebbsc(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(*)

```

### 5.8.67. 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(*)

```

### 5.8.68. 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(*)

```

### 5.8.69. 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(*)
  integer(4) :: copyValues, baseIdx
  character(c_char), device :: pBuffer(*)

```

### 5.8.70. `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
```

### 5.8.71. `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
```

### 5.8.72. `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
```

### 5.8.73. `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
```

### 5.8.74. `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(*)
```

### 5.8.75. `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
  type(cusparseMatDescr) :: descrC
  integer(4), device :: bsrRowPtrC(*), bsrColIndC(*)
  integer(4) :: rowBlockDimC, colBlockDimC
  character(c_char), device :: pBuffer(*)
```

### 5.8.76. `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(*)
```



### 5.8.77. `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(*)
```

### 5.8.78. `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(*)
```

### 5.8.79. `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(*)
```

### 5.8.80. `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
  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(*)
```

### 5.8.81. `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(*)
```

### 5.8.82. `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(*)
```

### 5.8.83. `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
```

### 5.8.84. 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
```

### 5.8.85. 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
```

### 5.8.86. 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(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  integer(4) :: rowBlockDim, colBlockDim
  integer(4) :: pBufferSize ! integer(8) also accepted
```

### 5.8.87. 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(*)
```

### 5.8.88. `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(*)
```

### 5.8.89. `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(*)
```

### 5.8.90. `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(*)
```

### 5.8.91. `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
  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(*)
```

### 5.8.92. `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(*)
```

### 5.8.93. `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
```

### 5.8.94. `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(*)
```

### 5.8.95. `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(*)
```

### 5.8.96. 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
```

### 5.8.97. 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(*)
```

### 5.8.98. 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
```

### 5.8.99. 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(*)
```

### 5.8.100. 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(cusparseCsr2csrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

### 5.8.101. `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(cusparseCsrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

### 5.8.102. `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(cusparseCsrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

### 5.8.103. `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(cusparseCsrInfo) :: info
  integer(8) :: pBufferSizeInBytes
```

### 5.8.104. `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(cusparseCsrInfo) :: info
  character(c_char), device :: pBuffer(*)
```

### 5.8.105. `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(cusparsCsrInfo) :: info
character(c_char), device :: pBuffer(*)

```

### 5.8.106. `cusparsCsr2csr`

This function transfers unsorted CSR format to CSR format.

```

integer(4) function cusparsCsr2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparsMatDescr) :: descrA
  complex(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparsCsrInfo) :: info
  character(c_char), device :: pBuffer(*)

```

### 5.8.107. `cusparsZcsr2csr`

This function transfers unsorted CSR format to CSR format.

```

integer(4) function cusparsZcsr2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparsMatDescr) :: descrA
  complex(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparsCsrInfo) :: info
  character(c_char), device :: pBuffer(*)

```

### 5.8.108. `cusparsScsr2csr`

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```

integer(4) function cusparsScsr2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparsMatDescr) :: descrA
  real(4), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparsCsrInfo) :: info
  character(c_char), device :: pBuffer(*)

```

### 5.8.109. `cusparsDcsr2csr`

This function performs the backwards transformation from sorted CSR format to unsorted CSR format.

```

integer(4) function cusparsDcsr2csr(handle, m, n, nnz, descrA, csrVal,
csrRowPtr, csrColInd, info, pBuffer)
  type(cusparsHandle) :: handle
  integer(4) :: m, n, nnz
  type(cusparsMatDescr) :: descrA
  real(8), device :: csrVal(*)
  integer(4), device :: csrRowPtr(*), csrColInd(*)
  type(cusparsCsrInfo) :: info
  character(c_char), device :: pBuffer(*)

```



### 5.8.110. `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(*)
```

### 5.8.111. `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(*)
```

## 5.9. 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.

### 5.9.1. `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
```

### 5.9.2. `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(*)
```

### 5.9.3. 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(*)
```

### 5.9.4. 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
```

### 5.9.5. 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(*)
```

### 5.9.6. 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
```

### 5.9.7. cusparseDestroySpVec

This function releases the host memory associated with the sparse vector descriptor used in the generic API.

```
integer(4) function cusparseDestroySpVec(descr)
```

```
type(cusparsSpVecDescr) :: descr
```

### 5.9.8. cusparsSpVecGet

This function returns the fields within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparsSpVecGet(descr, size, nnz, indices, values,
  idxType, idxBase, valueType)
  type(cusparsSpVecDescr) :: descr
  integer(8) :: size, nnz
  type(c_devpnr) :: indices
  type(c_devpnr) :: values
  integer(4) :: idxType, idxBase, valueType
```

### 5.9.9. cusparsSpVecGetIndexBase

This function returns "idxBase" field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparsSpVecGetIndexBase(descr, idxBase)
  type(cusparsSpVecDescr) :: descr
  integer(4) :: idxBase
```

### 5.9.10. cusparsSpVecGetValues

This function returns the "values" field within the sparse vector descriptor used in the generic API.

```
integer(4) function cusparsSpVecGetValues(descr, values)
  type(cusparsSpVecDescr) :: descr
  type(c_devpnr) :: values
```

### 5.9.11. cusparsSpVecSetValues

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 cusparsSpVecSetValues(descr, values)
  type(cusparsSpVecDescr) :: descr
  real(4), device :: values(*)
```

### 5.9.12. cusparsCreateDnVec

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 cusparsCreateDnVec(descr, size, values, valueType)
  type(cusparsDnVecDescr) :: descr
  integer(8) :: size
  real(4), device :: values(*)
  integer(4) :: valueType
```

### 5.9.13. 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
```

### 5.9.14. 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
```

### 5.9.15. 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
```

### 5.9.16. 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(*)
```

### 5.9.17. 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
```

### 5.9.18. 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
```

### 5.9.19. 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
```

### 5.9.20. 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
```

### 5.9.21. 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
```

### 5.9.22. 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
  integer(8) :: rows, cols, nnz
  type(c_devptr) :: cooRowInd, cooColInd, cooValues
  integer(4) :: idxType, idxBase, valueType
```

### 5.9.23. 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
```

### 5.9.24. 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
```

### 5.9.25. 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
```

### 5.9.26. 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(*)
```

### 5.9.27. 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(*)
```

### 5.9.28. 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
```

### 5.9.29. 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
```

### 5.9.30. 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
```

### 5.9.31. 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
```

### 5.9.32. 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(*)
```

### 5.9.33. 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
```

### 5.9.34. cusparseSpMatSetStridedBatch

This function sets the "batchCount" field within the sparse matrix descriptor used in the generic API.

```
integer(4) function cusparseSpMatSetStridedBatch(descr, batchCount)
  type(cusparseSpMatDescr) :: descr
  integer(4) :: batchCount
```

### 5.9.35. 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
```

### 5.9.36. 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
```

### 5.9.37. 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
```



### 5.9.38. `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
```

### 5.9.39. `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_devp_ptr) :: values
  integer(4) :: valueType, order
```

### 5.9.40. `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_devp_ptr) :: values
```

### 5.9.41. `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(*)
```

### 5.9.42. `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
```

### 5.9.43. `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
```

### 5.9.44. `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
  real(4), device :: result ! device or host variable
  integer(4) :: computeType
  integer(8), intent(out) :: bufferSize
```

### 5.9.45. `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(*)
```

### 5.9.46. `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
```

### 5.9.47. `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
real(4), device :: alpha, beta ! device or host variable
type(cusparsespmatDescr) :: matA
type(cusparsednvecDescr) :: vecX, vecY
integer(4) :: computeType, alg
integer(4), device :: buffer(*)
```

### 5.9.48. 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
```

### 5.9.49. 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
```

### 5.9.50. 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(cusparsespsvHandle) :: 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
```

### 5.9.51. 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(cusparsespsvHandle) :: 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(*)
```

### 5.9.52. cusparsespsv\_solve

This function executes the solve phase for the sparse triangular linear system.

```
integer(4) function cusparsespsv_solve(handle, opA, alpha, matA, vecX, vecY,
&
```

```

computeType, alg, spsvDescr)
type(cusparsHandle) :: handle
integer(4) :: opA
real(4) :: alpha ! device or host variable
type(cusparsSpMatDescr) :: matA
type(cusparsDnVecDescr) :: vecX, vecY
integer(4) :: computeType, alg
type(cusparsSpSVDScr) :: spsvDescr

```

### 5.9.53. cusparsSpMM\_bufferSize

This function returns the size of the workspace needed by `cusparsSpMM()`. The value returned is in bytes.

```

integer(4) function cusparsSpMM_bufferSize(handle, opA, opB, alpha, matA,
matB, beta, matC, computeType, alg, bufferSize)
type(cusparsHandle) :: handle
integer(4) :: opA, opB
real(4), device :: alpha, beta ! device or host variable
type(cusparsSpMatDescr) :: matA
type(cusparsDnMatDescr) :: matB, matC
integer(4) :: computeType, alg
integer(8), intent(out) :: bufferSize

```

### 5.9.54. cusparsSpMM\_preprocess

This function can be used to speedup the `cusparsSpMM` computation. See the CUSPARSE Library documentation for the capabilities supported in each release.

```

integer(4) function cusparsSpMM_preprocess(handle, opA, opB, alpha, matA,
matB, beta, matC, computeType, alg, buffer)
type(cusparsHandle) :: handle
integer(4) :: opA, opB
real(4), device :: alpha, beta ! device or host variable
type(cusparsSpMatDescr) :: matA
type(cusparsDnMatDescr) :: matB, matC
integer(4) :: computeType, alg
integer(4), device :: buffer(*)

```

### 5.9.55. cusparsSpMM

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 `cusparsSpMM_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 cusparsSpMM(handle, opA, opB, alpha, matA, matB, beta,
matC, computeType, alg, buffer)
type(cusparsHandle) :: handle
integer(4) :: opA, opB
real(4), device :: alpha, beta ! device or host variable
type(cusparsSpMatDescr) :: matA
type(cusparsDnMatDescr) :: matB, matC
integer(4) :: computeType, alg
integer(4), device :: buffer(*)

```

### 5.9.56. 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
```

### 5.9.57. 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
```

### 5.9.58. 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
```

### 5.9.59. 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(*)
```

### 5.9.60. cusparseSpSM\_solve

This function executes the solve phase for the sparse triangular linear system.

```
integer(4) function cusparseSpSM_solve(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
```

### 5.9.61. `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
```

### 5.9.62. `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(*)
```

### 5.9.63. `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(*)
```

### 5.9.64. `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
```

### 5.9.65. `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
```

### 5.9.66. `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 `cusparseSpGEMM_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
  type(cusparseSpMatDescr) :: matA, MatB, MatC
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize1
  integer(4), device :: buffer1
```

### 5.9.67. `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 `cusparseSpGEMM_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
  integer(4) :: computeType, alg
  type(cusparseSpGEMMDescr) :: spgemmDescr
  integer(8) :: bufferSize2
  integer(4), device :: buffer2
```

### 5.9.68. `cusparseSpGEMM_copy`

This function, along with `cusparseSpGEMM_workEstimation()` and `cusparseSpGEMM_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(cusparsHandle) :: handle
integer(4) :: opA, opB
real(4) :: alpha, beta ! device or host variable
type(cusparsSpMatDescr) :: matA, MatB, MatC
integer(4) :: computeType, alg
type(cusparsSpGEMMDescr) :: 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 = cusparsCreateCsr(matC, nd, nd, 0, nullp, nullp, &
    nullp, CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I, &
    CUSPARSE_INDEX_BASE_ONE, CUDA_R_64F)
call printStatus('cusparsCreateCsr', status, CUSPARSE_STATUS_SUCCESS)

status = cusparsSpGEMM_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('cusparsSpGEMM_workEstimation', status,
    CUSPARSE_STATUS_SUCCESS)

print *, "SpGEMM buffersize1 required: ", bsize
if (bsize.gt.0) allocate(buffer_d(bsize))

status = cusparsSpGEMM_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('cusparsSpGEMM_workEstimation', status,
    CUSPARSE_STATUS_SUCCESS)

status = cusparsSpGEMM_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('cusparsSpGEMM_compute', status, CUSPARSE_STATUS_SUCCESS)

print *, "SpGEMM buffersize2 required: ", bsize2
if (bsize2.gt.0) allocate(buffer2_d(bsize2))

status = cusparsSpGEMM_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('cusparsSpGEMM_compute', status, CUSPARSE_STATUS_SUCCESS)

status = cusparsSpMatGetSize(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 = cusparsCsrSetPointers(matC, csrRowPtrC_d, csrColIndC_d, csrValDC_d)
call printStatus('cusparsCsrSetPointers', status, CUSPARSE_STATUS_SUCCESS)

status = cusparsSpGEMM_copy(h, CUSPARSE_OPERATION_NON_TRANSPOSE, &
    CUSPARSE_OPERATION_NON_TRANSPOSE, Dalpha, matA, matB, Dbeta, &
    matC, CUDA_R_64F, CUSPARSE_SPGEMM_DEFAULT, spgemmd)
call printStatus('cusparsSpGEMM_copy', status, CUSPARSE_STATUS_SUCCESS)

if (bsize.gt.0) deallocate(buffer_d)
if (bsize2.gt.0) deallocate(buffer2_d)

```

### 5.9.69. cusparsSpGEMMreuse\_workEstimation

This function, along with **cusparsSpGEMM\_compute()**, are both used for determining the buffer requirements and for performing the actual computation. In the typical usage, the first call to **cusparsSpGEMM\_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
```

### 5.9.70. `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
  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
```

### 5.9.71. `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
```

## 5.9.72. `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 `cusparseSpGEMM_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 6.

## 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).

### 6.1. 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 = 1
integer, parameter :: CUSOLVER_VER_PATCH = 0
```

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
  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

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

```

```

enum, bind(c)
  enumerator :: CUSOLVERDN_GETRF = 0
end enum

```

### 6.1.1. 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

```

### 6.1.2. cusolverDnDestroy

This function releases CPU-side resources used by the cuSolverDn library.

```

integer(4) function cusolverDnDestroy(handle)
  type(cusolverDnHandle) :: handle

```

### 6.1.3. cusolverDnCreateParams

This function creates and initializes the 64-bit API structure to default values.

```

integer(4) function cusolverDnCreateParams(params)
  type(cusolverDnParams) :: params

```

### 6.1.4. cusolverDnDestroyParams

This function releases any resources used by the 64-bit API structure.

```

integer(4) function cusolverDnDestroyParams(params)

```

```
type(cusolverDnParams) :: params
```

### 6.1.5. 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
```

### 6.1.6. 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
```

### 6.1.7. 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
```

## 6.2. 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.

### 6.2.1. 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
```

### 6.2.2. 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
```

### 6.2.3. 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
```

### 6.2.4. 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
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

### 6.2.5. 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
```

### 6.2.6. 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
```

### 6.2.7. 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

```

## 6.2.8. 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

```

## 6.2.9. 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

```

## 6.2.10. 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

```

## 6.2.11. 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
```

## 6.2.12. 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
```

## 6.2.13. 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_devpstr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

## 6.2.14. 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_devpstr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

## 6.2.15. 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_devpstr), device :: Aarray(*)
    integer(4), device, intent(out) :: devinfo
```

## 6.2.16. 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_devp_ptr), device :: Aarray(*)
integer(4), device, intent(out) :: devinfo

```

## 6.2.17. 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
integer(4) :: n, nrhs, lda, ldb, batchCount
type(c_devp_ptr), device :: Aarray(*)
type(c_devp_ptr), device :: Barray(*)
integer(4), device, intent(out) :: devinfo

```

## 6.2.18. 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_devp_ptr), device :: Aarray(*)
type(c_devp_ptr), device :: Barray(*)
integer(4), device, intent(out) :: devinfo

```

## 6.2.19. 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_devp_ptr), device :: Aarray(*)
type(c_devp_ptr), device :: Barray(*)
integer(4), device, intent(out) :: devinfo

```

## 6.2.20. 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
```

### 6.2.21. 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
```

### 6.2.22. 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
```

### 6.2.23. 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
```

### 6.2.24. 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
  integer(4) :: n, lda
  complex(8), device, dimension(lda,*) :: A
  integer(4) :: lwork
```

## 6.2.25. 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
```

## 6.2.26. 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
```

## 6.2.27. 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
```

## 6.2.28. 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
```

## 6.2.29. 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
```

## 6.2.30. 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
```

## 6.2.31. 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
```

## 6.2.32. 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
```

## 6.2.33. 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
```

### 6.2.34. 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
```

### 6.2.35. 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
```

### 6.2.36. 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
```

### 6.2.37. 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
```

## 6.2.38. 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
```

## 6.2.39. 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
```

## 6.2.40. 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
    integer(4) :: n, lda
    complex(8), device, dimension(lda,*) :: A
    integer(4) :: lwork
```

## 6.2.41. 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
```

## 6.2.42. 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

```

### 6.2.43. 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

```

### 6.2.44. 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

```

### 6.2.45. 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

```

### 6.2.46. 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

```



## 6.2.47. 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
```

## 6.2.48. 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
```

## 6.2.49. 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
```

## 6.2.50. 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
```

## 6.2.51. 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
  integer(4), device, dimension(*) :: devipiv
```

```
integer(4), device, intent(out) :: devinfo
```

## 6.2.52. 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
```

## 6.2.53. 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
```

## 6.2.54. 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
```

## 6.2.55. 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
```

## 6.2.56. 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
```

## 6.2.57. 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
```

## 6.2.58. 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
```

## 6.2.59. 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
```

## 6.2.60. 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
```

## 6.2.61. 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
```

## 6.2.62. 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
```

## 6.2.63. 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
```

## 6.2.64. 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
```

## 6.2.65. 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
```

## 6.2.66. cusolverDnDgeqrf

This function computes the QR factorization of an  $m \times n$  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
```

## 6.2.67. cusolverDnCgeqrf

This function computes the QR factorization of an  $m \times n$  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
```

## 6.2.68. cusolverDnZgeqrf

This function computes the QR factorization of an  $m \times n$  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
```

## 6.2.69. 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
    real(4), device, dimension(lda,*) :: A
    real(4), device, dimension(*) :: tau
    integer(4) :: lwork
```

## 6.2.70. 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
```

## 6.2.71. 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
```

## 6.2.72. 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
```

## 6.2.73. 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
```

## 6.2.74. 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
```

## 6.2.75. 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

```

## 6.2.76. 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
    complex(8), device, dimension(*) :: tau
    complex(8), device, dimension(lwork) :: workspace
    integer(4), device, intent(out) :: devinfo

```

## 6.2.77. 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

```

## 6.2.78. 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

```

## 6.2.79. 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
```

## 6.2.80. 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
```

## 6.2.81. 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
```

## 6.2.82. 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
```

## 6.2.83. 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
```

## 6.2.84. cusolverDnZormqr

This function generates the unitary matrix  $Q$  from the QR factorization of an  $m \times n$  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
```

## 6.2.85. 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
```

## 6.2.86. 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
```

## 6.2.87. 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
```

## 6.2.88. 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

```

## 6.2.89. cusolverDnSsytrf

This function computes the Bunch-Kaufman factorization of an  $n \times n$  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
real(4), device, dimension(lwork) :: workspace
integer(4), device, intent(out) :: devinfo

```

## 6.2.90. cusolverDnDsytrf

This function computes the Bunch-Kaufman factorization of an  $n \times n$  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

```

## 6.2.91. cusolverDnCsytrf

This function computes the Bunch-Kaufman factorization of an  $n \times n$  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

```

## 6.2.92. cusolverDnZsytrf

This function computes the Bunch-Kaufman factorization of an  $n \times n$  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
```

### 6.2.93. 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
```

### 6.2.94. 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
```

### 6.2.95. 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
```

### 6.2.96. 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
```

## 6.2.97. cusolverDnSsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an  $n \times n$  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
```

## 6.2.98. cusolverDnDsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an  $n \times n$  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
```

## 6.2.99. cusolverDnCsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an  $n \times n$  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
```

## 6.2.100. cusolverDnZsytrs

This function solves the system of linear equations resulting from the Bunch-Kaufman factorization of an  $n \times n$  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

```

### 6.2.101. 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

```

### 6.2.102. 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

```

### 6.2.103. 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

```

### 6.2.104. 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

```

## 6.2.105. 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
```

## 6.2.106. 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
```

## 6.2.107. 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
```

## 6.2.108. 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
```

## 6.3. cusolverDn Legacy Eigenvalue Solver API

This section describes the eigenvalue solver legacy API of cusolverDn, including bidiagonalization and SVD.

### 6.3.1. 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
```

### 6.3.2. 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
```

### 6.3.3. 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
```

### 6.3.4. 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
```

### 6.3.5. 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

```

### 6.3.6. cusolverDnDgebrd

This function reduces a general  $m \times n$  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
  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

```

### 6.3.7. cusolverDnCgebrd

This function reduces a general  $m \times n$  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

```

### 6.3.8. cusolverDnZgebrd

This function reduces a general  $m \times n$  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

```



### 6.3.9. 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
```

### 6.3.10. 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
```

### 6.3.11. 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
```

### 6.3.12. 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
```

### 6.3.13. 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
```

### 6.3.14. 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
```

### 6.3.15. 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
```

### 6.3.16. 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
```

### 6.3.17. 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
```

### 6.3.18. 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
```

### 6.3.19. 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
```

### 6.3.20. 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
```

### 6.3.21. cusolverDnSsytrd

This function reduces a general symmetric or Hermitian  $n \times n$  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
```

### 6.3.22. cusolverDnDsytrd

This function reduces a general symmetric or Hermitian  $n \times n$  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
```

### 6.3.23. cusolverDnChetrd

This function reduces a general symmetric or Hermitian  $n \times n$  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
```

### 6.3.24. cusolverDnZhetrd

This function reduces a general symmetric or Hermitian  $n \times n$  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

```

### 6.3.25. 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

```

### 6.3.26. 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

```

### 6.3.27. 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

```

### 6.3.28. 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

```

### 6.3.29. 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

```

### 6.3.30. 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

```

### 6.3.31. 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

```

### 6.3.32. 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

```

### 6.3.33. 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

```

### 6.3.34. 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

```

### 6.3.35. 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

```

### 6.3.36. 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
```

### 6.3.37. 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
```

### 6.3.38. 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
```

### 6.3.39. 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
```

### 6.3.40. 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
```



### 6.3.41. 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
```

### 6.3.42. 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
```

### 6.3.43. 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
```

### 6.3.44. 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
```

### 6.3.45. cusolverDnSgesvd

This function computes the singular value decomposition (SVD) of an  $m \times n$  matrix  $A$  and the corresponding left and/or right singular vectors. CusolverDnSgesvd only supports  $m \geq n$ .

```
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
```

```
real(4), device, dimension(lwork) :: rwork
integer(4), device, intent(out) :: devinfo
```

### 6.3.46. cusolverDnDgesvd

This function computes the singular value decomposition (SVD) of an  $m \times n$  matrix  $A$  and the corresponding left and/or right singular vectors. `CusolverDnDgesvd` only supports  $m \geq n$ .

```
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
```

### 6.3.47. cusolverDnCgesvd

This function computes the singular value decomposition (SVD) of an  $m \times n$  matrix  $A$  and the corresponding left and/or right singular vectors. `CusolverDnCgesvd` only supports  $m \geq n$ .

```
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
```

### 6.3.48. cusolverDnZgesvd

This function computes the singular value decomposition (SVD) of an  $m \times n$  matrix  $A$  and the corresponding left and/or right singular vectors. `CusolverDnZgesvd` only supports  $m \geq n$ .

```
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
    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
```

### 6.3.49. 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
```

### 6.3.50. 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
```

### 6.3.51. 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
```

### 6.3.52. 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
```

### 6.3.53. cusolverDnSsyevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.54. cusolverDnDsyevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.55. cusolverDnCHeevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.56. cusolverDnZheevd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.57. 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
```

### 6.3.58. 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
```

### 6.3.59. 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
```

### 6.3.60. 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
```

### 6.3.61. cusolverDnSsyevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.62. cusolverDnDsyevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.63. cusolverDnCheevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.64. cusolverDnZheevdx

This function computes all or a selection of eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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

```

### 6.3.65. 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

```

### 6.3.66. 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

```

### 6.3.67. 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

```

### 6.3.68. 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
```

### 6.3.69. 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
```

### 6.3.70. 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
```

### 6.3.71. 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
```

### 6.3.72. cusolverDnZhegvd

This function computes eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.73. 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
```

### 6.3.74. 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
```

### 6.3.75. 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

```

### 6.3.76. 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

```

### 6.3.77. 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

```

### 6.3.78. 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

```

### 6.3.79. cusolverDnChegvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
```

### 6.3.80. cusolverDnZhegvdx

This function computes all or a selection of the eigenvalues and eigenvectors of a symmetric or Hermitian  $n \times n$  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
    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
```

## 6.4. 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.

### 6.4.1. 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

```

## 6.4.2. 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
    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

```

## 6.4.3. 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

```

## 6.4.4. 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
```

### 6.4.5. 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
```

### 6.4.6. 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
```

### 6.4.7. 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
```

### 6.4.8. cusolverDnXgetrs

This function solves the linear system of multiple right-hand sides applying a general  $n \times n$  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
```

### 6.4.9. 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

```

### 6.4.10. cusolverDnXsyevd

This function computes the eigenvalues and eigenvectors of a symmetric (Hermitian)  $n \times n$  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

```

### 6.4.11. 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

```

## 6.4.12. cusolverDnXsyevdx

This function computes all or some of the eigenvalues and optionally the eigenvectors of a symmetric (Hermitian)  $n \times n$  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
```

## 6.4.13. 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
```



## 6.4.14. cusolverDnXgesvd

This function computes the singular value decomposition (SVD) of an  $m \times n$  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
```

## 6.4.15. 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
```

## 6.4.16. cusolverDnXgesvdp

This function computes the singular value decomposition (SVD) of an  $m \times n$  matrix. This variation combines polar decomposition 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
```

## 6.5. 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.

### 6.5.1. 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, :))
  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.

## 6.5.2. 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,:))
    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`.

## 6.5.3. 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
  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**.

## 6.5.4. 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**.

### 6.5.5. 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	

### 6.5.6. 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)
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 NV_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 **NV\_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 7.

## 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 v1.0.0, and an experimental high-level module which maps several standard Fortran intrinsic functions to the functionality contained within the cuTENSOR library.

Chapter 10 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
```

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)`.

### 7.1. CUTENSOR Definitions and Helper Functions

This section contains definitions and data types used in the cuTENSOR library and interfaces to the cuTENSOR helper functions.

The cuTENSOR module contains the following derived type definitions:

```
! Definitions from cutensor.h
integer, parameter :: CUTENSOR_MAJOR = 1
integer, parameter :: CUTENSOR_MINOR = 1
integer, parameter :: CUTENSOR_PATCH = 0

! Types from cutensor/types.h
```



```

! Algorithm Control
type, bind(c) :: cutensorAlgo
  integer(4) :: algo
end type
type(cutensorAlgo), parameter :: &
  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_RECOMMENDED = 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
  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.
  ! 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
! 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)

```

```

! Compute Type
type, bind(c) :: cutensorComputeType
  integer(4) :: ctyp
end type
type(cutensorComputeType), parameter :: &
  CUTENSOR_R_MIN_16F = cutensorComputeType(2**0), & ! real as a half
  CUTENSOR_C_MIN_16F = cutensorComputeType(2**1), & ! complex as a half
  CUTENSOR_R_MIN_32F = cutensorComputeType(2**2), & ! real as a float
  CUTENSOR_C_MIN_32F = cutensorComputeType(2**3), & ! complex as a float
  CUTENSOR_R_MIN_64F = cutensorComputeType(2**4), & ! real as a double
  CUTENSOR_C_MIN_64F = cutensorComputeType(2**5), & ! complex as a double
  CUTENSOR_R_MIN_8U = cutensorComputeType(2**6), & ! real as a uint8
  CUTENSOR_R_MIN_32U = cutensorComputeType(2**7), & ! real as a uint32
  CUTENSOR_R_MIN_8I = cutensorComputeType(2**8), & ! real as a int8
  CUTENSOR_R_MIN_32I = cutensorComputeType(2**9), & ! real as a int32
  CUTENSOR_R_MIN_16BF = cutensorComputeType(2**10), & ! real as a bfloat16
  CUTENSOR_R_MIN_TF32 = cutensorComputeType(2**11), & ! real as a tf32
  CUTENSOR_C_MIN_TF32 = cutensorComputeType(2**12) ! complex as a tf32

```

```

! cuTENSOR descriptors and other types to hold state

```

```

type cutensorHandle
  integer(8) :: fields(512)
end type

```

```

type cutensorDescriptor
  integer(8) :: fields(64)
end type

```

```

type cutensorContractionDescriptor
  integer(8) :: fields(256)
end type

```

```

type cutensorContractionPlan
  integer(8) :: fields(640)
end type

```

```

type cutensorContractionFind
  integer(8) :: fields(64)

```

```

end type

! CUDA Datatype whose values are common among libraries
type, bind(c) :: cudaDataType
  integer(4) :: ctyp
end type

```

### 7.1.1. cutensorInit

This function initializes the cuTENSOR library and returns a handle for subsequent cuTENSOR calls.

```

type(cutensorStatus) function cutensorInit(handle)
  type(cutensorHandle) :: handle

```

### 7.1.2. cutensorInitTensorDescriptor

This function initializes a cuTENSOR descriptor, given the number of modes, extents, strides, and type of the data.

```

type(cutensorStatus) function cutensorInitTensorDescriptor(handle, desc,
  numModes, extent, stride, dataType, unaryOp)
  type(cutensorHandle) :: handle
  type(cutensorDescriptor) :: desc
  integer(4) :: numModes
  integer(8), dimension(*) :: extent
  integer(8), dimension(*) :: stride
  type(cudaDataType) :: dataType
  type(cutensorOperator) :: unaryOp

```

Alternatively, cuTENSOR will take a value of NULL for the stride, and assume a packed array. You can pass `c_null_ptr` (from the `iso_c_binding` module) for the stride in that case.

### 7.1.3. cutensorGetAlignmentRequirement

This function computes the minimal alignment requirement for a given data pointer and descriptor. The `ptr` argument can be any type, kind, or rank, but must be device data.

```

type(cutensorStatus) function cutensorGetAlignmentRequirement(handle, ptr, desc,
  alignment)
  type(cutensorHandle) :: handle
  real, device, dimension(*) :: ptr
  type(cutensorDescriptor) :: desc
  integer(4), intent(out) :: alignment

```

### 7.1.4. cutensorGetErrorString

This function returns the description string for an error code.

```

character*128 function cutensorGetErrorString(ierr)
  type(cutensorStatus) :: ierr

```

### 7.1.5. cutensorGetVersion

This function returns the version number of the cuTENSOR library.

```

integer(8) function cutensorGetVersion()

```

## 7.1.6. cutensorGetCudartVersion

This function returns the version of the CUDA runtime that the cuTENSOR library was compiled against.

```
integer(8) function cutensorGetCudartVersion()
```

## 7.2. CUTENSOR Element-wise Operations

This section contains interfaces for the cuTENSOR functions that perform element-wise operations between tensors.

### 7.2.1. cutensorPermutation

This function performs an out-of-place tensor permutation of the form

**B=alpha\*op (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 operations `op(perm(A))` are set up in the call to **cutensorInitTensorDescriptor ()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorPermutation(handle, &
  alpha, A, descA, modeA, B, descB, modeB, typeScalar, stream)
  type(cutensorHandle) :: handle
  type(cutensorDescriptor) :: descA, descB
  real, device, dimension(*) :: A, B
  real :: alpha
  integer(4), dimension(*) :: modeA, modeB
  type(cudaDataType) :: typeScalar
  integer(kind=cuda_stream_kind) :: stream
```

### 7.2.2. cutensorElementwiseBinary

This function performs an element-wise tensor operation on two inputs of the form

**D=opAC (alpha\*op (perm (A) ) , gamma\*op (perm (C) ) )**

The **opAC** argument is an element-wise binary operator. The type and kind of the **alpha**, **gamma** scalars is determined by the `typeScalar` argument. 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. The operations `op(perm(A))`, `op(perm(C))`, etc. are set up in the call to **cutensorInitTensorDescriptor ()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorElementwiseBinary(handle, &
  alpha, A, descA, modeA, gamma, C, descC, modeC, &
  D, descD, modeD, opAC, typeScalar, stream)
  type(cutensorHandle) :: handle
  type(cutensorDescriptor) :: descA, descC, descD
  real, device, dimension(*) :: A, C, D
  real :: alpha, gamma
  integer(4), dimension(*) :: modeA, modeC, modeD
  type(cutensorOperator) :: opAC
  type(cudaDataType) :: typeScalar
  integer(kind=cuda_stream_kind) :: stream
```

### 7.2.3. cutensorElementwiseTrinary

This function performs an element-wise tensor operation on three inputs of the form

**$D = \text{opABC}(\text{opAB}(\alpha * \text{op}(\text{perm}(A))), \text{beta} * \text{op}(\text{perm}(B))), \text{gamma} * \text{op}(\text{perm}(C)))$**

The **opABC**, **opAB** arguments are element-wise binary operators. The type and kind of the **alpha**, **beta**, **gamma** scalars is determined by the **typeScalar** argument. 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. The operations **op(perm(A))**, **op(perm(B))**, etc. are set up in the call to **cutensorInitTensorDescriptor()** via the **unaryOp** argument.

```
type(cutensorStatus) function cutensorElementwiseTrinary(handle, &
  alpha, A, descA, modeA, beta, B, descB, modeB, gamma, C, descC, modeC, &
  D, descD, modeD, opAB, opABC, typeScalar, stream)
type(cutensorHandle) :: handle
type(cutensorDescriptor) :: descA, descB, descC, descD
real, device, dimension(*) :: A, B, C, D
real :: alpha, beta, gamma
integer(4), dimension(*) :: modeA, modeB, modeC, modeD
type(cutensorOperator) :: opAB, opABC
type(cudaDataType) :: typeScalar
integer(kind=cuda_stream_kind) :: stream
```

## 7.3. CUTENSOR Reduction Operations

This section contains interfaces for the cuTENSOR functions that perform reduction operations on tensors.

### 7.3.1. cutensorReductionGetWorkspace

This function determines the workspace needed for a given tensor reduction performed by **cutensorReduction()**.

```
type(cutensorStatus) function cutensorReductionGetWorkspace(handle, &
  A, descA, modeA, C, descC, modeC, D, descD, modeD, &
  opReduce, minTypeCompute, workspaceSize)

type(cutensorHandle) :: handle
real, device, dimension(*) :: A, C, D
type(cutensorDescriptor) :: descA, descC, descD
integer(4), dimension(*) :: modeA, modeC, modeD
type(cutensorOperator) :: opReduce
type(cutensorComputeType) :: minTypeCompute
integer(8), intent(out) :: workspaceSize
```

### 7.3.2. cutensorReduction

This function performs a tensor reduction of the form  **$D = \alpha * \text{opReduce}(\text{op}(A)) + \text{beta} * \text{op}(C)$** . The **opReduce** argument controls what type of reduction is performed. The arrays **A**, **C**, and **D** can be of any supported type, kind, and rank. The **alpha** and **beta** scalars' type and kind is determined by the **minTypeCompute** argument.

```
type(cutensorStatus) function cutensorReduction(handle, &
  alpha, A, descA, modeA, beta, C, descC, modeC, D, descD, modeD, &
  opReduce, minTypeCompute, workspace, workspaceSize, stream)
```

```

type(cutensorHandle) :: handle
real, device, dimension(*) :: A, C, D
type(cutensorDescriptor) :: descA, descC, descD
integer(4), dimension(*) :: modeA, modeC, modeD
real :: alpha, beta
type(cutensorOperator) :: opReduce
type(cutensorComputeType) :: minTypeCompute
real, device, dimension(*) :: workspace
integer(8) :: workspaceSize
integer(kind=cuda_stream_kind) :: stream

```

## 7.4. CUTENSOR Contraction Operations

This section contains interfaces for the cuTENSOR functions that perform contraction operations between tensors.

### 7.4.1. cutensorInitContractionDescriptor

This function initializes the contraction descriptor for a contraction problem of the form

$$\mathbf{D} = \alpha \mathbf{A} \mathbf{B} + \beta \mathbf{C}$$

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. The strides and extents of the arrays **A**, **B**, **C**, **D** are set up in the array descriptors by calls to **cutensorInitTensorDescriptor()**. The alignment requirements for each array can be found by calls to **cutensorGetAlignmentRequirement**.

```

type(cutensorStatus) function cutensorInitContractionDescriptor(handle, desc, &
  descA, modeA, algnA, descB, modeB, algnB, &
  descC, modeC, algnC, descD, modeD, algnD, computeType)
type(cutensorHandle) :: handle
type(cutensorContractionDescriptor) :: desc
integer(4), dimension(*) :: modeA, modeB, modeC, modeD
integer(4) :: algnA, algnB, algnC, algnD
type(cutensorComputeType) :: computeType

```

### 7.4.2. cutensorInitContractionFind

This function limits the search space of viable algorithms for contraction operations.

```

type(cutensorStatus) function cutensorInitContractionFind(handle, &
  find, algo)
type(cutensorHandle) :: handle
type(cutensorContractionFind) :: find
type(cutensorAlgo) :: algo

```

### 7.4.3. cutensorInitContractionPlan

This function initializes the contraction plan for a tensor contraction operation.

```

type(cutensorStatus) function cutensorInitContractionPlan(handle, &
  plan, desc, find, workspaceize)
type(cutensorHandle) :: handle
type(cutensorContractionPlan) :: plan
type(cutensorContractionDescriptor) :: desc
type(cutensorContractionFind) :: find
integer(8) :: workspaceSize

```

### 7.4.4. cutensorContractionGetWorkspace

This function determines the workspace needed for a given tensor contraction performed by `cutensorContraction()`.

```
type(cutensorStatus) function cutensorContractionGetWorkspace(handle, &
  desc, find, pref, workspaceSize)
  type(cutensorHandle) :: handle
  type(cutensorContractionDescriptor) :: desc
  type(cutensorContractionFind) :: find
  type(cutensorWorkspacePreference) :: pref
  integer(8), intent(out) :: workspaceSize
```

### 7.4.5. cutensorContractionMaxAlgos

This function returns the maximum number of algorithms available to perform tensor contractions.

```
type(cutensorStatus) function cutensorContractionMaxAlgos(maxNumAlgos)
  integer(4), intent(out) :: maxNumAlgos
```

### 7.4.6. cutensorContraction

This function performs a tensor contraction of the form

$$\mathbf{D} = \mathbf{alpha} * (\mathbf{A} * \mathbf{B}) + \mathbf{beta} * \mathbf{C}$$

The arrays A, B, C, and D can be of any supported type, kind, and rank. The alpha and beta scalars type and kind is determined by the `typeCompute` argument given to `cutensorInitContractionDescriptor()` and encoded in the plan.

```
type(cutensorStatus) function cutensorContraction(handle, &
  plan, alpha, A, B, beta, C, D, workspace, workspaceSize, stream)
  type(cutensorHandle) :: handle
  type(cutensorContractionPlan) :: plan
  real, device, dimension(*) :: A, B, C, D
  real :: alpha, beta
  real, device, dimension(*) :: workspace
  integer(8) :: workspaceSize
  integer(kind=cuda_stream_kind) :: stream
```

## 7.5. 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
```

### 7.5.1. 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]))
```

### 7.5.2. Fortran Transpose

This Fortran function transposes 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(transpose(A))
```

### 7.5.3. 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))
```

## 7.5.4. 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
```

## 7.5.5. 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 the compute type `CUTENSOR_R_MIN_TF32` rather than `CUTENSOR_R_MIN_32F` 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 `CUTENSOR_R_MIN_TF32`.

```
integer function cutensorExSetOptLevel(level)
integer(4) :: level
```

## 7.5.6. 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 8.

## 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)`.

### 8.1. 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 = 7
integer, parameter :: NCCL_PATCH = 3

! 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
```

```

! 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), &
  ncclNumResults       = ncclResult(6)

! ncclDataType
type, bind(c) :: ncclDataType
  integer(c_int) :: member
end type ncclDataType

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), &
  ncclNumTypes  = ncclDataType(9)

! 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), &
  ncclNumOps   = ncclRedOp(4)

```

### 8.1.1. ncclGetVersion

This function returns the version number of the NCCL library.

```

type(ncclResult) function ncclGetVersion(version)
integer(4) :: version

```

### 8.1.2. 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

```

### 8.1.3. ncclCommInitRank

This function generates a new NCCL communicator, of type(ncclComm). The rank argument must be between 0 and n ranks-1. The uniqueId argument should be generated with ncclGetUniqueId.

```
type(ncclResult) function ncclCommInitRank(comm, n ranks, uniqueId, rank)
type(ncclComm) :: comm
integer(4) :: n ranks
type(ncclUniqueId) :: uniqueId
integer(4) :: rank
```

### 8.1.4. ncclCommInitAll

This function creates a single-process communicator clique, an array of type(ncclComm).

```
type(ncclResult) function ncclCommInitAll(comms, n dev, devlist)
type(ncclComm) :: comms(*)
integer(4) :: n dev
integer(4) :: devlist(*)
```

### 8.1.5. ncclCommDestroy

This function frees resources allocated to a NCCL communicator. It will wait for uncompleted operations.

```
type(ncclResult) function ncclCommDestroy(comm)
type(ncclComm) :: comm
```

### 8.1.6. ncclCommAbort

This function frees resources allocated to a NCCL communicator. It will abort uncompleted operations.

```
type(ncclResult) function ncclCommAbort(comm)
type(ncclComm) :: comm
```

### 8.1.7. ncclGetErrorString

This function returns an error string for a given ncclResult value.

```
character*128 function ncclGetErrorString(ierr)
type(ncclResult) :: ierr
```

### 8.1.8. ncclCommGetAsyncError

This function queries whether the communicator has encountered any asynchronous errors.

```
type(ncclResult) function ncclCommGetAsyncError(comm, asyncError)
type(ncclComm) :: comm
type(ncclResult) :: asyncError
```

### 8.1.9. 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
```

### 8.1.10. 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
```

### 8.1.11. 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
```

## 8.2. 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.

### 8.2.1. 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
```

## 8.2.2. 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_devp_ptr) :: 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

```

## 8.2.3. 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_devp_ptr) :: 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

```

## 8.2.4. 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_devp_ptr) :: 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

```



## 8.2.5. 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_devpstr) :: 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

```

## 8.3. 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\_devpstr) arguments. The point to point operations were added in NCCL 2.7.

### 8.3.1. 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_devpstr) :: 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

```

### 8.3.2. 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_devpstr) :: 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

```

## 8.4. NCCL Group Calls

This section contains interfaces for the NCCL functions that begin and end a group such that multiple calls can be merged.

### 8.4.1. 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()
```

### 8.4.2. 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()
```

# Chapter 9.

## 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).

## 9.1. 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_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

! 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

! 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)
```

### 9.1.1. nvshmem\_init

This subroutine allocates and initializes resources used by the NVSHMEM library.

```
subroutine nvshmem_init()
```

### 9.1.2. 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(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, nranks, ierr)
!
nvstat = nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, MPI_COMM_WORLD)
```

### 9.1.3. 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()
```

### 9.1.4. nvshmem\_n\_pes

This function returns the number of PEs running in the program.

```
integer(4) function nvshmem_n_pes()
```

### 9.1.5. 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
```

### 9.1.6. 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
```

### 9.1.7. 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
```

### 9.1.8. 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
```

### 9.1.9. 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
```

### 9.1.10. 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
```

### 9.1.11. nvshmem\_team\_destroy

This function is a collective operation which destroys the team and frees the resources associated with it.

```
integer(4) function nvshmem_team_destroy(team)
    integer(4) :: team
```

### 9.1.12. 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
```

### 9.1.13. 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
```

### 9.1.14. nvshmem\_finalize

This subroutine releases resources and ends the NVSHMEM portion of a program started with `nvshmem_init()`.

```
subroutine nvshmem_finalize()
```

### 9.1.15. 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_ptr_i(dest, pe)
  integer :: dest ! Any kind and rank
  integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptr_l(dest, pe)
  logical :: dest ! Any kind and rank
  integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptr_r(dest, pe)
  real :: dest ! Any kind and rank
  integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptr_c(dest, pe)
  complex :: dest ! Any kind and rank
  integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptr_c1(dest, pe)
  character :: dest ! Any kind and rank
  integer(4) :: pe
```

```
type(c_devptr) function nvshmem_ptr_cd(dest, pe)
  type(c_devptr) :: dest
  integer(4) :: pe
```

## 9.2. NVSHMEM Memory Management Functions

This section contains the Fortran interfaces to NVSHMEM functions used to manage the symmetric heap.

### 9.2.1. 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)
```

### 9.2.2. 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.
```

### 9.2.3. 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
```

### 9.2.4. 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
```

## 9.3. 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).

### 9.3.1. `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_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put16(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put16_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put32(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put32_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put64(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put64_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put128(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put128_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

```

### 9.3.2. 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
  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

```

### 9.3.3. 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_devpnr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iput8_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devpnr) :: 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_devpnr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iput16_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devpnr) :: 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_devpnr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iput32_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devpnr) :: 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_devpnr) :: 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
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

```

### 9.3.4. 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
  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_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put32_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put64_nbi(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put64_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_put128_nbi(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_put128_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

```

### 9.3.5. 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_devp_ptr) :: 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

```

### 9.3.6. 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_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

```
subroutine nvshmemx_put16_warp(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

```
subroutine nvshmemx_put32_warp(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

```
subroutine nvshmemx_put64_warp(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

```
subroutine nvshmemx_put128_warp(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

### 9.3.7. 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_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
```

```
subroutine nvshmemx_getmem_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: 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
```

```

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

```

### 9.3.8. 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
```

### 9.3.9. 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_devp_ptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iget8_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devp_ptr) :: 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_devp_ptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iget16_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devp_ptr) :: 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_devp_ptr) :: dest, source
integer(8) :: dst, sst, nelems
integer(4) :: pe

```

```

subroutine nvshmemx_iget32_on_stream(dest, source, dst, sst, nelems, pe, stream)
type(c_devp_ptr) :: 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_devp_ptr) :: 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

```

### 9.3.10. 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(*)
  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_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_get32_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_get64_nbi(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_get64_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

subroutine nvshmem_get128_nbi(dest, source, nelems, pe)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe

subroutine nvshmemx_get128_nbi_on_stream(dest, source, nelems, pe, stream)
  type(c_devp_ptr) :: dest, source
  integer(8) :: nelems
  integer(4) :: pe
  integer(cuda_stream_kind) :: stream

```

### 9.3.11. 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_devp_ptr) :: 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

```

### 9.3.12. 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
```

## 9.4. 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.

### 9.4.1. `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
```

## 9.4.2. 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
```

## 9.4.3. 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_int8_alltoall(team, dest, source, nelems)
  integer(4) :: team
  integer(1), device :: dest, source
  integer(8) :: nelems

integer function nvshmem_int16_alltoall(team, dest, source, nelems)
  integer(4) :: team
  integer(2), device :: dest, source
  integer(8) :: nelems

integer function nvshmem_int32_alltoall(team, dest, source, nelems)
  integer(4) :: team
  integer(4), device :: dest, source
  integer(8) :: nelems

integer function nvshmem_int64_alltoall(team, dest, source, nelems)
  integer(4) :: team
  integer(8), device :: dest, source
  integer(8) :: nelems

integer function nvshmem_float_alltoall(team, dest, source, nelems)
  integer(4) :: team
  real(4), device :: dest, source
  integer(8) :: nelems

integer function nvshmem_double_alltoall(team, dest, source, nelems)
  integer(4) :: team
  real(8), device :: dest, source
```



```

integer(8) :: nelems

integer function nvshmemx_int8_alltoall_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_alltoall_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_alltoall_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_alltoall_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_alltoall_on_stream(team, &
    dest, source, nelems, stream)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems
    integer(cuda_stream_kind) :: stream

integer function nvshmemx_double_alltoall_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_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    integer(1), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int16_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    integer(2), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int32_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    integer(4), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_int64_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    integer(8), device :: dest, source
    integer(8) :: nelems

integer function nvshmemx_float_alltoall_block(team, dest, source, nelems)
    integer(4) :: team
    real(4), device :: dest, source
    integer(8) :: nelems

```

```

integer function nvshmemx_double_alltoall_block(team, dest, source, nelems)
  integer(4) :: team
  real(8), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_int8_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  integer(1), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_int16_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  integer(2), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_int32_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  integer(4), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_int64_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  integer(8), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_float_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  real(4), device :: dest, source
  integer(8) :: nelems

```

```

integer function nvshmemx_double_alltoall_warp(team, dest, source, nelems)
  integer(4) :: team
  real(8), device :: dest, source
  integer(8) :: nelems

```

#### 9.4.4. 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)
  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
    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

```

### 9.4.5. 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
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
  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
  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

```

## 9.4.6. 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.

### 9.4.6.1. 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
  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)
  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
```

#### 9.4.6.2. 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)
  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

```

### 9.4.6.3. 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
```

#### 9.4.6.4. 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)
  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
    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)
    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

```

#### 9.4.6.5. 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
```

```
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

```

#### 9.4.6.6. 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
  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
  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

```

#### 9.4.6.7. 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
  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)
  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
end function

integer function nvshmemx_double_prod_reduce_block(team, dest, source, nreduce)
  integer(4) :: team
  real(8), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_int8_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  integer(1), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_int16_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  integer(2), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_int32_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  integer(4), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_int64_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  integer(8), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_float_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  real(4), device :: dest, source
  integer(8) :: nreduce
end function

integer function nvshmemx_double_prod_reduce_warp(team, dest, source, nreduce)
  integer(4) :: team
  real(8), device :: dest, source
  integer(8) :: nreduce
end function

```

## 9.5. 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.

### 9.5.1. `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.

```

subroutine nvshmem_int32_wait_until(ivar, cmp, value)
  integer(4), device :: ivar
  integer(4) :: cmp, value
end subroutine

subroutine nvshmemx_int32_wait_until_on_stream(ivar, cmp, value, stream)
  integer(4), device :: ivar
end subroutine

```

```

integer(4) :: cmp, value
integer(cuda_stream_kind) :: stream

subroutine nvshmem_int64_wait_until(ivar, cmp, value)
integer(8), device :: ivar
integer(4) :: cmp
integer(8) :: value

subroutine nvshmemx_int64_wait_until_on_stream(ivar, cmp, value, stream)
integer(8), device :: ivar
integer(4) :: cmp
integer(8) :: value
integer(cuda_stream_kind) :: stream

```

## 9.6. 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.

### 9.6.1. nvshmem\_fence

This subroutine ensures the ordering of delivery of operations on symmetric data objects.

```
subroutine nvshmem_fence()
```

### 9.6.2. 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

```

# Chapter 10.

## 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 **-lnvhpcwrapnvtx** to their link line.

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)`.

### 10.1. 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.

### 10.1.1. 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.

```
subroutine nvtxStartRange( label, icolor )
  character(len=*) :: label
  integer, optional :: icolor
```

### 10.1.2. nvtxEndRange

This subroutine terminates a simple labelled time span range initiated by **nvtxStartRange**. It takes no arguments.

```
subroutine nvtxEndRange()
```

## 10.2. NVTX Advanced Tooling APIs

This section describes the advanced Fortran interfaces to the NVIDIA Tools Extension (NVTX) library which target the NVTX v3 API.

### 10.2.1. 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.

```
! Parameters
integer, parameter :: NVTX_VERSION = 3
integer, parameter :: NVTX_EVENT_ATTRIB_STRUCT_SIZE = 48

! NVTX Status
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

! nvtxCOLORType_t, from nvToolsExt.h
type, bind(c) :: nvtxCOLORType
  integer(4) :: type
end type
type(nvtxCOLORType), parameter :: &
  NVTX_COLOR_UNKNOWN = nvtxCOLORType(0), &
  NVTX_COLOR_ARGB = nvtxCOLORType(1)

! nvtxMessageType_t, from nvToolsExt.h
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)

! nvtxPayloadType_t, from nvToolsExt.h
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)

! Something just for Fortran ease of use, C compat.
! The Fortran structure is bigger, but the first 48 bytes are the same
! Making it allocatable means it will get deallocated properly
type nvtxFtnStringType
  character(1), allocatable :: chars(:)
end type

! nvtxEventAttributes_v2, from nvToolsExt.h
type, bind(C) :: nvtxEventAttributes
  integer(C_INT16_T)      :: version = NVTX_VERSION
  integer(C_INT16_T)      :: size = NVTX_EVENT_ATTRIB_STRUCT_SIZE
  integer(C_INT)          :: category = 0
  type(nvtxColorType)     :: colorType = NVTX_COLOR_ARGB
  integer(C_INT)          :: color = z'ffffffff'
  type(nvtxPayloadType)   :: payloadType = NVTX_PAYLOAD_UNKNOWN
  integer(C_INT)          :: reserved0
  integer(C_INT64_T)      :: payload ! union uint,int,double
  type(nvtxMessageType)  :: messageType = NVTX_MESSAGE_TYPE_ASCII
  type(nvtxFtnStringType) :: message ! ascii char
end type

! This module provides a type constructor for the nvtxEventAttributes type.
! For example:
! event = nvtxEventAttributes(message, color)
! message can be a Fortran character string, or
! an nvtx registered string.
! color is an optional argument, integer(C_INT), assigned to
! the color field

type nvtxRangeId
  integer(8) :: id
end type

type nvtxDomainHandle
  type(C_PTR) :: handle
end type

type nvtxStringHandle
  type(C_PTR) :: handle
end type

```

## 10.2.2. 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()

```



### 10.2.3. 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
```

### 10.2.4. nvtxDomainDestroy

This subroutine destroys an NVTX domain.

```
subroutine nvtxDomainDestroy(domain)
  type(nvtxDomainHandle) :: domain
```

### 10.2.5. nvtxDomainRegisterString

This function registers an immutable string with NVTX, for use with the **type (eventAttributes)** message field.

```
function nvtxDomainRegisterString(domain, message) &
  result(stringHandle)
  type(nvtxDomainHandle) :: domain
  character(len=*) :: 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 (eventAttributes)** 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)
```

### 10.2.6. 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
```

### 10.2.7. nvtxNameCategory

This subroutine allows the user to assign a name to a category ID.

```
subroutine nvtxNameCategory(category, name)
  integer(4) :: category
  character(len=*) :: name
```

### 10.2.8. 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
```

### 10.2.9. nvtxMarkEx

This subroutine marks an instantaneous event in the application, with user-supplied NVTX event attributes.

```
subroutine nvtxMarkEx(event)
  type(nvtxEventAttributes) :: event
```

### 10.2.10. nvtxMark

This subroutine marks an instantaneous event in the application with a user-supplied message.

```
subroutine nvtxMark(message)
  character(len=*) :: message
```

### 10.2.11. 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
```

### 10.2.12. 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
```

### 10.2.13. 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
```

## 10.2.14. 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
```

## 10.2.15. 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
```

## 10.2.16. 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
```

## 10.2.17. 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
```

## 10.2.18. 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
```

## 10.2.19. nvtxDomainRangePop

This functions ends a nested thread range in the application, within a specific domain.

```
function nvtxDomainRangePop(domain) result(ilvl)
  type(nvtxDomainHandle) :: domain
  integer(4) :: ilvl
```

## 10.2.20. 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
```

## 10.3. 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.

# Chapter 11.

## EXAMPLES

This section contains examples with source code.

### 11.1. 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)
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
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)
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

```

## 11.2. 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
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
!
! 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)
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

```

## 11.3. 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)), &
                                             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

```

## 11.4. 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

  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

```

## 11.5. 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
  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
    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
      end do
      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
    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 separate devices
status = cufftXtMalloc(planR2C, phi_desc, CUFFT_XT_FORMAT_INPLACE)
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
            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, j1, 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 j1 = 1, nyl
                        j = j1 + yOffset
                        do i = 1, nx/2+1
                            k2 = kx_d(i)**2 + ky_d(j)**2 + kz_d(k)**2
                            phi_d(i,j1,k) = -phi_d(i,j1,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
    end do
end block

```

```

do g = 1, nGPUs
  dev = phiXtDesc%GPUs(g)
  status = cudaSetDevice(dev)
  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

```

## 11.6. 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,
! 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 kernel 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
  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_dptra(:, :, :)
  type(c_ptr) :: tmpcptra
#else
  complex, pointer, device :: u_dptra(:, :, :)
#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;
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

! 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')

call checkCufft(cufftMakePlan3d(planr2c, nz, ny, nx, CUFFT_R2C, &
  worksizes), 'cufftMakePlan3d r2c error')
call checkCufft(cufftMakePlan3d(planc2r, nz, ny, nx, CUFFT_C2R, &
  worksizes), '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

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 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
    tmpcptr = transfer(u_descptr%data(1), tmpcptr)
    call c_f_pointer(tmpcptr, u_dptra, local_permuted_cshape(1:3))

    call scalingData(local_permuted_cshape(1), local_permuted_cshape(2), &
        local_permuted_cshape(3), u_dptra, real(nx*ny*nz))
#else
    call c_f_pointer(u_descptr%data(1), u_dptra, 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_dptra(i,j,k) = u_dptra(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

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx inverse
call checkCufft(cufftXtExecDescriptor(planr2r, u_desc, u_desc, &
    CUFFT_INVERSE), 'inverse fft failed')

! These are equivalent, and work as well
! istat = cufftXtMemcpy(planr2r, 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(planr2r))

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
    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, nz
            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
        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
        do j = 1, ny
            do i = 1, nz
                data(i, j, k) = data(i, j, k) / factor
            end do
        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
    !@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

```

## 11.7. 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 curl(1000, .true.); call curl(1000, .false.)
call cur2(1000, .true.); call cur2(1000, .false.)
call cur3(1000, .true.); call cur3(1000, .false.)
end
!
subroutine curl(n, onhost)
use curand
integer :: a(n)
type(curandGenerator) :: g
integer(8) nbits
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
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"
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
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

```

## 11.8. 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"
  end if
end program t

```

```

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 program

```

## 11.9. 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)
  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
end subroutine

```

```

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)
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

```

## 11.10. 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
  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

```

```

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

```

## 11.11. 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)

! 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

! initialize 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
! 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

```

## 11.12. 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

    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

! initialize 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
! 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

```



## 11.13. Using cuTENSOR from CUDA Fortran Host Code

This example demonstrates the use of the low-level cuTENSOR module 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 testcutensorcufl
use cudafor
use cutensor

integer, parameter :: ndim = 3
real, managed, allocatable :: dA(:,:,:), dC(:,:)
real, allocatable :: hA(:,:,:), hC(:,:)
real, device, allocatable :: workbuf(:)
real :: alpha, beta
integer(4) :: modesA, modesC
integer(8), dimension(ndim) :: extA, strA, extC, strC
integer(4), dimension(ndim) :: modeA, modeC
integer(8) :: workbufsize

type(cutensorStatus) :: cstat
type(cutensorHandle) :: handle
type(cutensorDescriptor) :: descA, descC

! Init
cstat = cutensorInit(handle)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(workbuf(2500))
workbufsize = 2500 * 4

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 ! Reshape Order Values
modesA = ndim

cstat = cutensorInitTensorDescriptor(handle, descA, modesA, extA, strA, &
                                     cudaDataType(CUDA_R_32F), CUTENSOR_OP_IDENTITY)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

allocate(dC(80,100))
dC = 0.0
extC(1:ndim-1) = shape(dC)
strC(1) = 1; strC(2) = 80
modeC(1) = 1; modeC(2) = 3 ! Skip mode 2 for reduction across that dim
modesC = ndim-1

cstat = cutensorInitTensorDescriptor(handle, descC, modesC, extC, strC, &
                                     cudaDataType(CUDA_R_32F), CUTENSOR_OP_IDENTITY)
if (cstat.ne.CUTENSOR_STATUS_SUCCESS) print *,cutensorGetErrorString(cstat)

```

```

alpha = 1.0; beta = 0.0
cstat = cutensorReduction(handle, alpha, dA, descA, modeA, &
    beta, dC, descC, modeC, dC, descC, modeC, &
    CUTENSOR_OP_ADD, CUTENSOR_R_MIN_32F, &
    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
end program

```

## 11.14. 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.

### 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
    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

```

## 11.15. 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
!$acc kernels
  do j = 1, nj
    do i = 1, ni
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
!$acc end kernels
end do
!$acc exit data copyout(d)

print *,sum(d)
end program
```

## Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

## 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 NVIDIA Corporation. All rights reserved.