cuSOLVER Library
Table of Contents

Chapter 1. Introduction........................................................................................................ 1
  1.1. cuSolverDN: Dense LAPACK........................................................................................ 2
  1.2. cuSolverSP: Sparse LAPACK ...................................................................................... 2
  1.3. cuSolverRF: Refactorization......................................................................................... 2
  1.4. Naming Conventions.................................................................................................... 3
  1.5. Asynchronous Execution............................................................................................. 4
  1.6. Library Property......................................................................................................... 5
  1.7. high precision package............................................................................................... 5

Chapter 2. Using the CUSOLVER API.................................................................................. 6
  2.1. General description...................................................................................................... 6
    2.1.1. Thread Safety.......................................................................................................... 6
    2.1.2. Scalar Parameters................................................................................................. 6
    2.1.3. Parallelism with Streams...................................................................................... 6
    2.1.4. Link Third-party LAPACK Library....................................................................... 6
    2.1.5. convention of info............................................................................................... 7
    2.1.6. usage of _bufferSize......................................................................................... 7
  2.2. cuSolver Types Reference.......................................................................................... 7
    2.2.1. cuSolverDN Types............................................................................................... 7
      2.2.1.1. cusolverDnHandle_t......................................................................................... 8
      2.2.1.2. cublasFillMode_t............................................................................................ 8
      2.2.1.3. cublasOperation_t.......................................................................................... 8
      2.2.1.4. cusolverEigType_t........................................................................................ 8
      2.2.1.5. cusolverEigMode_t......................................................................................... 8
      2.2.1.6. cusolverIRSRefinement_t............................................................................ 9
      2.2.1.7. cusolverDnIRSParams_t............................................................................... 10
      2.2.1.8. cusolverDnIRSInfos_t.................................................................................. 10
      2.2.1.9. cusolverDnFunction_t.................................................................................... 10
      2.2.1.10. cusolverAlgMode_t...................................................................................... 10
      2.2.1.11. cusolverStatus_t......................................................................................... 10
    2.2.2. cuSolverSP Types............................................................................................... 10
      2.2.2.1. cusolverSpHandle_t....................................................................................... 11
      2.2.2.2. cusparseMatDescr_t..................................................................................... 11
      2.2.2.3. cusolverStatus_t.......................................................................................... 11
    2.2.3. cuSolverRF Types............................................................................................... 12
      2.2.3.1. cusolverRfHandle_t...................................................................................... 12
2.2.3.2. cusolverRfMatrixFormat_t.................................................................12
2.2.3.3. cusolverRfNumericBoostReport_t..................................................13
2.2.3.4. cusolverRfResetValuesFastMode_t..................................................13
2.2.3.5. cusolverRfFactorization_t...............................................................13
2.2.3.6. cusolverRfTriangularSolve_t..........................................................13
2.2.3.7. cusolverRfUnitDiagonal_t...............................................................13
2.2.3.8. cusolverStatus_t.............................................................................14

2.3. cuSolver Formats Reference.....................................................................14
2.3.1. Index Base Format...............................................................................14
2.3.2. Vector [Dense] Format.........................................................................14
2.3.3. Matrix [Dense] Format.........................................................................14
2.3.4. Matrix [CSR] Format...........................................................................15
2.3.5. Matrix [CSC] Format...........................................................................16

2.4. cuSolverDN: dense LAPACK Function Reference.....................................17
2.4.1. cuSolverDN Helper Function Reference.............................................17
2.4.1.1. cusolverDnCreate()..........................................................................17
2.4.1.2. cusolverDnDestroy().........................................................................17
2.4.1.3. cusolverDnSetStream().....................................................................18
2.4.1.4. cusolverDnGetStream()....................................................................18
2.4.1.5. cusolverDnCreateSyevjInfo().............................................................18
2.4.1.6. cusolverDnDestroySyevjInfo()..........................................................19
2.4.1.7. cusolverDnXsyevjSetTolerance().........................................................19
2.4.1.8. cusolverDnXsyevjSetMaxSweeps()......................................................19
2.4.1.9. cusolverDnXsyevjSetSortEig().............................................................19
2.4.1.10. cusolverDnXsyevjGetResidual().........................................................20
2.4.1.11. cusolverDnXsyevjGetSweeps().........................................................20
2.4.1.12. cusolverDnCreateGesvdjInfo()........................................................21
2.4.1.13. cusolverDnDestroyGesvdjInfo().......................................................21
2.4.1.14. cusolverDnXgesvdjSetTolerance()......................................................21
2.4.1.15. cusolverDnXgesvdjSetMaxSweeps()..................................................22
2.4.1.16. cusolverDnXgesvdjSetSortEig().........................................................22
2.4.1.17. cusolverDnXgesvdjGetResidual()......................................................22
2.4.1.18. cusolverDnXgesvdjGetSweeps().........................................................23
2.4.1.19. cusolverDnIRSParamsCreate().........................................................23
2.4.1.20. cusolverDnIRSParamsDestroy().......................................................24
2.4.1.21. cusolverDnIRSParamsSetSolverPrecisions()......................................24
2.4.1.22. cusolverDnIRSParamsSetSolverMainPrecision()..............................25
2.4.1.23. cusolverDnIRSParamsSetSolverLowestPrecision().........................26
2.4.1.24. cusolverDnIRSParamsSetRefinementSolver() ........................................................................................27
2.4.1.25. cusolverDnIRSParamsSetTol() .................................................................................................................28
2.4.1.26. cusolverDnIRSParamsSetTolInner() ...........................................................................................................29
2.4.1.27. cusolverDnIRSParamsSetMaxIters() ...........................................................................................................29
2.4.1.28. cusolverDnIRSParamsSetMaxItersInner() .............................................................................................. 30
2.4.1.29. cusolverDnIRSParamsEnableFallback() ................................................................................................. 30
2.4.1.30. cusolverDnIRSParamsDisableFallback() .......................................................................................... 31
2.4.1.31. cusolverDnIRSParamsGetMaxIters() ...........................................................................................................31
2.4.1.32. cusolverDnIRSInfosCreate() .....................................................................................................................32
2.4.1.33. cusolverDnIRSInfosDestroy() ...................................................................................................................33
2.4.1.34. cusolverDnIRSInfosGetMaxIters() ............................................................................................................33
2.4.1.35. cusolverDnIRSInfosGetNiters() ................................................................................................................33
2.4.1.36. cusolverDnIRSInfosGetOuterNiters() .........................................................................................................34
2.4.1.37. cusolverDnIRSInfosRequestResidual() .................................................................................................34
2.4.1.38. cusolverDnIRSInfosGetResidualHistory() .............................................................................................. 35
2.4.1.39. cusolverDnCreateParams() .......................................................................................................................36
2.4.1.40. cusolverDnDestroyParams() ....................................................................................................................36
2.4.1.41. cusolverDnSetAdvOptions() .....................................................................................................................37

2.4.2. Dense Linear Solver Reference (legacy) ...........................................................................................................37
2.4.2.1. cusolverDn<t>potrf() ................................................................................................................................. 37
2.4.2.2. cusolverDnPotrf() [[DEPRECATED]] ........................................................................................................ 40
2.4.2.3. cusolverDn<t>potrs() ............................................................................................................................... 42
2.4.2.4. cusolverDnPotrs() [[DEPRECATED]] ......................................................................................................... 44
2.4.2.5. cusolverDn<t>potri() ...................................................................................................................................46
2.4.2.6. cusolverDn<t>getrf() ...................................................................................................................................48
2.4.2.7. cusolverDnGetrf() [[DEPRECATED]] ....................................................................................................... 51
2.4.2.8. cusolverDn<t>getrs() ...................................................................................................................................53
2.4.2.9. cusolverDnGetrs() [[DEPRECATED]] ........................................................................................................ 55
2.4.2.10. cusolverDn<t1><t2>gesv() ................................................................................................................................57
2.4.2.11. cusolverDnIRSXgesv() ........................................................................................................................... 70
2.4.2.12. cusolverDn<t>geqrf() ...................................................................................................................................75
2.4.2.13. cusolverDnGeqrf() [[DEPRECATED]] .......................................................................................................... 78
2.4.2.14. cusolverDn<t1><t2>gels() .....................................................................................................................................80
2.4.2.15. cusolverDnIRSXgels() ............................................................................................................................. 93
2.4.2.16. cusolverDn<t>ormqr() ..................................................................................................................................98
2.4.2.17. cusolverDn<t>orgqr() ................................................................................................................................102
2.4.2.18. cusolverDn<t>sytrf() ....................................................................................................................................104
2.4.2.19. cusolverDn<t>potrfBatched() ..................................................................................................................107
2.4.2.20. `cusolverDn<t>potrsBatched()` ................................................................. 109
2.4.3. Dense Eigenvalue Solver Reference (legacy) .................................................. 111
  2.4.3.1. `cusolverDn<t>gebrd()` ............................................................................ 111
  2.4.3.2. `cusolverDn<t>orgbr()` ............................................................................ 114
  2.4.3.3. `cusolverDn<t>sytrd()` ............................................................................. 117
  2.4.3.4. `cusolverDn<t>ormtr()` ............................................................................ 121
  2.4.3.5. `cusolverDn<t>orgtr()` ............................................................................. 125
  2.4.3.6. `cusolverDn<t>gesvd()` ........................................................................... 127
  2.4.3.7. `cusolverDnGesvd() [DEPRECATED]` .......................................................... 131
  2.4.3.8. `cusolverDn<t>gesvdj()` .......................................................................... 134
  2.4.3.9. `cusolverDn<t>gesvdjBatched()` ............................................................ 139
  2.4.3.10. `cusolverDn<t>gesvdaStridedBatched()` ................................................ 143
  2.4.3.11. `cusolverDn<t>syevd()` ......................................................................... 149
  2.4.3.12. `cusolverDnSyevd() [DEPRECATED]` ....................................................... 152
  2.4.3.13. `cusolverDn<t>syevdx()` ........................................................................ 155
  2.4.3.14. `cusolverDnSyevdx() [DEPRECATED]` ..................................................... 160
  2.4.3.15. `cusolverDn<t>sgvvd()` .......................................................................... 164
  2.4.3.16. `cusolverDn<t>sgvvd()` .......................................................................... 164
  2.4.3.17. `cusolverDn<t>sgvvd()` .......................................................................... 174
  2.4.3.18. `cusolverDn<t>sgvvd()` .......................................................................... 178
  2.4.3.19. `cusolverDn<t>sgvvd()` .......................................................................... 183
2.4.4. Dense Linear Solver Reference (64-bit API) ..................................................... 187
  2.4.4.1. `cusolverDnXpotrf()` ............................................................................. 187
  2.4.4.2. `cusolverDnXpotrs()` ............................................................................. 190
  2.4.4.3. `cusolverDnXgetrf()` ............................................................................. 191
  2.4.4.4. `cusolverDnXgetrs()` ............................................................................. 194
  2.4.4.5. `cusolverDnXgeqrf()` ............................................................................. 196
2.4.5. Dense Eigenvalue Solver Reference (64-bit API) .............................................. 198
  2.4.5.1. `cusolverDnXgesvd()` ........................................................................... 198
  2.4.5.2. `cusolverDnXgesvd()` ........................................................................... 201
  2.4.5.3. `cusolverDnXsyevd()` ........................................................................... 204
  2.4.5.4. `cusolverDnXsyevdx()` ........................................................................... 207
2.5. `cuSolverSP`: sparse LAPACK Function Reference ............................................ 211
  2.5.1. Helper Function Reference .......................................................................... 211
    2.5.1.1. `cusolverSpCreate()` ........................................................................... 211
    2.5.1.2. `cusolverSpDestroy()` ........................................................................ 212
    2.5.1.3. `cusolverSpSetStream()` ..................................................................... 212
    2.5.1.4. `cusolverSpXcsrissym()` ..................................................................... 212
2.5.2. High Level Function Reference ........................................................................... 214
  2.5.2.1. cusolverSp<t>csrLsvlu()..................................................................................... 214
  2.5.2.2. cusolverSp<t>csrLsvqr().................................................................................. 217
  2.5.2.3. cusolverSp<t>csrLsvchol()............................................................................... 219
  2.5.2.4. cusolverSp<t>csrLsqvqr()................................................................................ 222
  2.5.2.5. cusolverSp<t>csreigvis().................................................................................. 225
  2.5.2.6. cusolverSp<t>csreigs()..................................................................................... 228

2.5.3. Low Level Function Reference ............................................................................ 231
  2.5.3.1. cusolverSpXcsrSymrcm().................................................................................. 231
  2.5.3.2. cusolverSpXcsrSymmdq()................................................................................ 232
  2.5.3.3. cusolverSpXcsrSymamd()............................................................................... 233
  2.5.3.4. cusolverSpXcsrMetisnd()................................................................................ 235
  2.5.3.5. cusolverSpXcsrZfd()....................................................................................... 236
  2.5.3.6. cusolverSpXcsrPerm()..................................................................................... 238
  2.5.3.7. cusolverSpXcsrQrBatched()............................................................................. 241

2.6. cuSolverRF: Refactorization Reference ................................................................ 246
  2.6.1. cusolverRfAccessBundledFactors().................................................................... 246
  2.6.2. cusolverRfAnalyze().......................................................................................... 247
  2.6.3. cusolverRfSetupDevice()................................................................................... 248
  2.6.4. cusolverRfSetupHost()....................................................................................... 250
  2.6.5. cusolverRfCreate()............................................................................................ 252
  2.6.6. cusolverRfExtractBundledFactorsHost()............................................................. 253
  2.6.7. cusolverRfExtractSplitFactorsHost()................................................................. 254
  2.6.8. cusolverRfDestroy().......................................................................................... 255
  2.6.9. cusolverRfGetMatrixFormat()........................................................................... 255
  2.6.10. cusolverRfGetNumericProperties()................................................................. 256
  2.6.11. cusolverRfGetNumericBoostReport().............................................................. 256
  2.6.12. cusolverRfGetResetValuesFastMode().............................................................. 257
  2.6.13. cusolverRfGetAlgs()....................................................................................... 257
  2.6.14. cusolverRfRefactor()....................................................................................... 257
  2.6.15. cusolverRfResetValues()................................................................................... 258
  2.6.16. cusolverRfSetMatrixFormat().......................................................................... 259
  2.6.17. cusolverRfSetNumericProperties()................................................................. 259
  2.6.18. cusolverRfSetResetValuesFastMode().............................................................. 260
  2.6.19. cusolverRfSetAlgs()........................................................................................ 261
  2.6.20. cusolverRfSolve()............................................................................................ 261
  2.6.21. cusolverRfBatchSetupHost()........................................................................... 262
  2.6.22. cusolverRfBatchAnalyze().............................................................................. 265
A.2. cuSolverRF-batch Example ................................................................. 296

Appendix B. CSR QR Batch Examples .................................................. 300
  B.1. Batched Sparse QR example 1 ....................................................... 300
  B.2. Batched Sparse QR example 2 ....................................................... 303

Appendix C. QR Examples ........................................................................ 308
  C.1. QR Factorization Dense Linear Solver ......................................... 308
  C.2. orthogonalization ........................................................................ 312

Appendix D. LU Examples ........................................................................ 316
  D.1. LU Factorization .......................................................................... 316

Appendix E. Cholesky Examples .............................................................. 321
  E.1. batched Cholesky Factorization .................................................... 321

Appendix F. Examples of Dense Eigenvalue Solver .................................. 325
  F.1. Standard Symmetric Dense Eigenvalue Solver ................................ 325
  F.2. Standard Symmetric Dense Eigenvalue Solver ................................ 328
  F.3. Generalized Symmetric-Definite Dense Eigenvalue Solver ............ 330
  F.4. Generalized Symmetric-Definite Dense Eigenvalue Solver ............ 334
  F.5. Standard Symmetric Dense Eigenvalue Solver (via Jacobi method) .... 337
  F.6. Generalized Symmetric-Definite Dense Eigenvalue Solver (via Jacobi method) ... 340
  F.7. batch eigenvalue solver for dense symmetric matrix ...................... 344

Appendix G. Examples of Singular Value Decomposition ....................... 350
  G.1. SVD with singular vectors ............................................................. 350
  G.2. SVD with singular vectors (via Jacobi method) .............................. 354
  G.3. batch dense SVD solver ............................................................... 358
  G.4. SVD approximation ..................................................................... 363
  G.5. 64-bit GESVD .............................................................................. 368
  G.6. 64-bit GESVDP ........................................................................... 372

Appendix H. Examples of multiGPU eigenvalue solver ............................ 377
  H.1. SYEVD of 1D Laplacian operator (example 1) .............................. 377
  H.2. SYEVD of 1D Laplacian operator (example 2) .............................. 388
  H.3. SYEVD of 1D Laplacian operator (example 3) .............................. 393

Appendix I. Examples of multiGPU linear solver .................................... 399
  I.1. GETRF and GETRS of 1D Laplacian operator (example 1) ............ 399
  I.2. POTRF and POTRS of 1D Laplacian operator (example 2) ............ 407
  I.3. POTRF and POTRI of 1D Laplacian operator (example 3) ............. 414

Appendix J. Acknowledgements ............................................................... 423

Appendix K. Bibliography ........................................................................ 425
List of Figures

Figure 1. Example of cusolveMG tiling for 3 Gpus ................................................................. 271
Figure 2. global matrix and local matrix .....................................................................................272
Chapter 1. Introduction

The cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries. It consists of two modules corresponding to two sets of API:

1. The cuSolver API on a single GPU
2. The cuSolverMG API on a single node multiGPU

Each of which can be used independently or in concert with other toolkit libraries. To simplify the notation, cuSolver denotes single GPU API and cuSolverMg denotes multiGPU API.

The intent of cuSolver is to provide useful LAPACK-like features, such as common matrix factorization and triangular solve routines for dense matrices, a sparse least-squares solver and an eigenvalue solver. In addition cuSolver provides a new refactorization library useful for solving sequences of matrices with a shared sparsity pattern.

cuSolver combines three separate components under a single umbrella. The first part of cuSolver is called cuSolverDN, and deals with dense matrix factorization and solve routines such as LU, QR, SVD and LDLT, as well as useful utilities such as matrix and vector permutations.

Next, cuSolverSP provides a new set of sparse routines based on a sparse QR factorization. Not all matrices have a good sparsity pattern for parallelism in factorization, so the cuSolverSP library also provides a CPU path to handle those sequential-like matrices. For those matrices with abundant parallelism, the GPU path will deliver higher performance. The library is designed to be called from C and C++.

The final part is cuSolverRF, a sparse re-factorization package that can provide very good performance when solving a sequence of matrices where only the coefficients are changed but the sparsity pattern remains the same.

The GPU path of the cuSolver library assumes data is already in the device memory. It is the responsibility of the developer to allocate memory and to copy data between GPU memory and CPU memory using standard CUDA runtime API routines, such as cudaMalloc(), cudaMemcpy(), and cudaMemcpyAsync().

cuSolverMg is GPU-accelerated ScaLAPACK. By now, cuSolverMg supports 1-D column block cyclic layout and provides symmetric eigenvalue solver.

Note: The cuSolver library requires hardware with a CUDA compute capability (CC) of at least 2.0 or higher. Please see the CUDA C++ Programming Guide for a list of the compute capabilities corresponding to all NVIDIA GPUs.
1.1. cuSolverDN: Dense LAPACK

The cuSolverDN library was designed to solve dense linear systems of the form

\[ Ax = b \]

where the coefficient matrix \( A \in \mathbb{R}^{m \times n} \), right-hand-side vector \( b \in \mathbb{R}^n \) and solution vector \( x \in \mathbb{R}^n \).

The cuSolverDN library provides QR factorization and LU with partial pivoting to handle a general matrix \( A \), which may be non-symmetric. Cholesky factorization is also provided for symmetric/Hermitian matrices. For symmetric indefinite matrices, we provide Bunch-Kaufman (LDL) factorization.

The cuSolverDN library also provides a helpful bidiagonalization routine and singular value decomposition (SVD).

The cuSolverDN library targets computationally-intensive and popular routines in LAPACK, and provides an API compatible with LAPACK. The user can accelerate these time-consuming routines with cuSolverDN and keep others in LAPACK without a major change to existing code.

1.2. cuSolverSP: Sparse LAPACK

The cuSolverSP library was mainly designed to solve sparse linear system

\[ Ax = b \]

and the least-squares problem

\[ x = \arg \min \| A z - b \| \]

where sparse matrix \( A \in \mathbb{R}^{m \times n} \), right-hand-side vector \( b \in \mathbb{R}^m \) and solution vector \( x \in \mathbb{R}^n \). For a linear system, we require \( m = n \).

The core algorithm is based on sparse QR factorization. The matrix \( A \) is accepted in CSR format. If matrix \( A \) is symmetric/Hermitian, the user has to provide a full matrix, ie fill missing lower or upper part.

If matrix \( A \) is symmetric positive definite and the user only needs to solve \( Ax = b \), Cholesky factorization can work and the user only needs to provide the lower triangular part of \( A \).

On top of the linear and least-squares solvers, the cuSolverSP library provides a simple eigenvalue solver based on shift-inverse power method, and a function to count the number of eigenvalues contained in a box in the complex plane.

1.3. cuSolverRF: Refactorization

The cuSolverRF library was designed to accelerate solution of sets of linear systems by fast re-factorization when given new coefficients in the same sparsity pattern.
where a sequence of coefficient matrices $A_i \in \mathbb{R}^{n \times n}$, right-hand-sides $f_i \in \mathbb{R}^n$ and solutions $x_i \in \mathbb{R}^n$ are given for $i=1,\ldots,k$.

The cuSolverRF library is applicable when the sparsity pattern of the coefficient matrices $A_i$ as well as the reordering to minimize fill-in and the pivoting used during the LU factorization remain the same across these linear systems. In that case, the first linear system ($i=1$) requires a full LU factorization, while the subsequent linear systems ($i=2,\ldots,k$) require only the LU re-factorization. The later can be performed using the cuSolverRF library.

Notice that because the sparsity pattern of the coefficient matrices, the reordering and pivoting remain the same, the sparsity pattern of the resulting triangular factors $L_i$ and $U_i$ also remains the same. Therefore, the real difference between the full LU factorization and LU re-factorization is that the required memory is known ahead of time.

1.4. Naming Conventions

The cuSolverDN library provides two different APIs; legacy and generic.

The functions in the legacy API are available for data types float, double, cuComplex, and cuDoubleComplex. The naming convention for the legacy API is as follows:

$$\text{cusolverDn}<t><\text{operation}>$$

where $<t>$ can be $S$, $D$, $C$, $Z$, or $X$, corresponding to the data types float, double, cuComplex, cuDoubleComplex, and the generic type, respectively. $<\text{operation}>$ can be Cholesky factorization ($\text{potrf}$), LU with partial pivoting ($\text{getrf}$), QR factorization ($\text{geqrf}$) and Bunch-Kaufman factorization ($\text{sytrf}$).

The functions in the generic API provide a single entry point for each routine and support for 64-bit integers to define matrix and vector dimensions. The naming convention for the generic API is data-agnostic and is as follows:

$$\text{cusolverDn}<\text{operation}>$$

where $<\text{operation}>$ can be Cholesky factorization ($\text{potrf}$), LU with partial pivoting ($\text{getrf}$) and QR factorization ($\text{geqrf}$).

The cuSolverSP library functions are available for data types float, double, cuComplex, and cuDoubleComplex. The naming convention is as follows:

$$\text{cusolverSp[Host]}<t>[<\text{matrix data format}>]<\text{operation}>[<\text{output matrix data format}>]<\text{based on}>$$

where $\text{cusolverSp}$ is the GPU path and $\text{cusolverSpHost}$ is the corresponding CPU path. $<t>$ can be $S$, $D$, $C$, $Z$, or $X$, corresponding to the data types float, double, cuComplex, cuDoubleComplex, and the generic type, respectively.

The $<\text{matrix data format}>$ is csr, compressed sparse row format.

The $<\text{operation}>$ can be $\text{ls}$, $\text{lsq}$, $\text{eig}$, $\text{eigs}$, corresponding to linear solver, least-square solver, eigenvalue solver and number of eigenvalues in a box, respectively.
The `<output matrix data format>` can be `v` or `m`, corresponding to a vector or a matrix. `<based on>` describes which algorithm is used. For example, `qr` (sparse QR factorization) is used in linear solver and least-square solver.

All of the functions have the return type `cusolverStatus_t` and are explained in more detail in the chapters that follow.

### cuSolverSP API

<table>
<thead>
<tr>
<th>routine</th>
<th>data format</th>
<th>operation</th>
<th>output format</th>
<th>based on</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>csrlsvlu</code></td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>LU (lu) with partial pivoting</td>
</tr>
<tr>
<td><code>csrlsvqr</code></td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>QR factorization (qr)</td>
</tr>
<tr>
<td><code>csrlsvchol</code></td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>Cholesky factorization (chol)</td>
</tr>
<tr>
<td><code>csrlsqvqr</code></td>
<td>csr</td>
<td>least-square solver (lsq)</td>
<td>vector (v)</td>
<td>QR factorization (qr)</td>
</tr>
<tr>
<td><code>csreigvsi</code></td>
<td>csr</td>
<td>eigenvalue solver (eig)</td>
<td>vector (v)</td>
<td>shift-inverse</td>
</tr>
<tr>
<td><code>csreigs</code></td>
<td>csr</td>
<td>number of eigenvalues in a box (eigs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>csrsymrcm</code></td>
<td>csr</td>
<td>Symmetric Reverse Cuthill-McKee (symrcm)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The cuSolverRF library routines are available for data type `double`. Most of the routines follow the naming convention:

```
cusolverRf_<operation>[(Host)][...]
```

where the trailing optional Host qualifier indicates the data is accessed on the host versus on the device, which is the default. The `<operation>` can be `Setup`, `Analyze`, `Refactor`, `Solve`, `ResetValues`, `AccessBundledFactors` and `ExtractSplitFactors`.

Finally, the return type of the cuSolverRF library routines is `cusolverStatus_t`.

### 1.5. Asynchronous Execution

The cuSolver library functions prefer to keep asynchronous execution as much as possible. Developers can always use the `cudaDeviceSynchronize()` function to ensure that the execution of a particular cuSolver library routine has completed.

A developer can also use the `cudaMemcpy()` routine to copy data from the device to the host and vice versa, using the `cudaMemcpyDeviceToHost` and
cudaMemcpyHostToDevice parameters, respectively. In this case there is no need to add a call to cudaMemcpyDeviceSynchronize() because the call to cudaMemcpy() with the above parameters is blocking and completes only when the results are ready on the host.

1.6. Library Property

The libraryPropertyType data type is an enumeration of library property types. (ie. CUDA version X.Y.Z would yield MAJOR_VERSION=X, MINOR_VERSION=Y, PATCH_LEVEL=Z)

```c
typedef enum libraryPropertyType_t
{
    MAJOR_VERSION,
    MINOR_VERSION,
    PATCH_LEVEL
} libraryPropertyType;
```

The following code can show the version of cusolver library.

```c
int major=-1,minor=-1,patch=-1;
cusolverGetProperty(MAJOR_VERSION, &major);
cusolverGetProperty(MINOR_VERSION, &minor);
cusolverGetProperty(PATCH_LEVEL, &patch);
printf("CUSOLVER Version (Major,Minor,PatchLevel): %d.%d.%d\n", major,minor,patch);
```

1.7. high precision package

The cusolver library uses high precision for iterative refinement when necessary.
Chapter 2. Using the CUSOLVER API

2.1. General description

This chapter describes how to use the cuSolver library API. It is not a reference for the cuSolver API data types and functions; that is provided in subsequent chapters.

2.1.1. Thread Safety

The library is thread-safe, and its functions can be called from multiple host threads.

2.1.2. Scalar Parameters

In the cuSolver API, the scalar parameters can be passed by reference on the host.

2.1.3. Parallelism with Streams

If the application performs several small independent computations, or if it makes data transfers in parallel with the computation, then CUDA streams can be used to overlap these tasks.

The application can conceptually associate a stream with each task. To achieve the overlap of computation between the tasks, the developer should:

1. Create CUDA streams using the function cudaStreamCreate(), and
2. Set the stream to be used by each individual cuSolver library routine by calling, for example, cusolverDnSetStream(), just prior to calling the actual cuSolverDN routine.

The computations performed in separate streams would then be overlapped automatically on the GPU, when possible. This approach is especially useful when the computation performed by a single task is relatively small, and is not enough to fill the GPU with work, or when there is a data transfer that can be performed in parallel with the computation.

2.1.4. Link Third-party LAPACK Library

Starting with CUDA 10.1 update 2, NVIDIA LAPACK library liblapack_static.a is a subset of LAPACK and only contains GPU accelerated stedc and bdsqr. The user has to link libcusolver_static.a with liblapack_static.a in order to build the application.
Using the CUSOLVER API

Successfully. Prior to CUDA 10.1 update 2, the user can replace liblapack_static.a with a third-party LAPACK library, for example, MKL. In CUDA 10.1 update 2, the third-party LAPACK library no longer affects the behavior of cusolver library, neither functionality nor performance. Furthermore the user cannot use liblapack_static.a as a standalone LAPACK library because it is only a subset of LAPACK.

Note: The liblapack_static.a library, which is the binary of CLAPACK-3.2.1, is a new feature of CUDA 10.0.

- If you use libcusolver_static.a, then you must link with liblapack_static.a explicitly, otherwise the linker will report missing symbols. No conflict of symbols between liblapack_static.a and other third-party LAPACK library, you are free to link the latter to your application.
- The liblapack_static.a is built inside libcusolver.so. Hence, if you use libcusolver.so, then you don’t need to specify a LAPACK library. The libcusolver.so will not pick up any routines from the third-party LAPACK library even you link the application with it.

2.1.5. convention of info

Each LAPACK routine returns an info which indicates the position of invalid parameter. If info = -i, then i-th parameter is invalid. To be consistent with base-1 in LAPACK, cusolver does not report invalid handle into info. Instead, cusolver returns CUSOLVER_STATUS_NOT_INITIALIZED for invalid handle.

2.1.6. usage of _bufferSize

There is no cudaMalloc inside cuSolver library, the user must allocate the device workspace explicitly. The routine xyz_bufferSize is to query the size of workspace of the routine xyz, for example xyz = potrf. To make the API simple, xyz_bufferSize follows almost the same signature of xyz even it only depends on some parameters, for example, device pointer is not used to decide the size of workspace. In most cases, xyz_bufferSize is called in the beginning before actual device data (pointing by a device pointer) is prepared or before the device pointer is allocated. In such case, the user can pass null pointer to xyz_bufferSize without breaking the functionality.

2.2. cuSolver Types Reference

2.2.1. cuSolverDN Types

The float, double, cuComplex, and cuDoubleComplex data types are supported. The first two are standard C data types, while the last two are exported from cuComplex.h. In addition, cuSolverDN uses some familiar types from cuBlas.
2.2.1.1. **cusolverDnHandle_t**

This is a pointer type to an opaque cuSolverDN context, which the user must initialize by calling `cusolverDnCreate()` prior to calling any other library function. An un-initialized Handle object will lead to unexpected behavior, including crashes of cuSolverDN. The handle created and returned by `cusolverDnCreate()` must be passed to every cuSolverDN function.

2.2.1.2. **cublasFillMode_t**

The type indicates which part (lower or upper) of the dense matrix was filled and consequently should be used by the function. Its values correspond to Fortran characters ‘L’ or ‘l’ (lower) and ‘U’ or ‘u’ (upper) that are often used as parameters to legacy BLAS implementations.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_FILL_MODE_LOWER</td>
<td>the lower part of the matrix is filled</td>
</tr>
<tr>
<td>CUBLAS_FILL_MODE_UPPER</td>
<td>the upper part of the matrix is filled</td>
</tr>
</tbody>
</table>

2.2.1.3. **cublasOperation_t**

The `cublasOperation_t` type indicates which operation needs to be performed with the dense matrix. Its values correspond to Fortran characters ‘N’ or ‘n’ (non-transpose), ‘T’ or ‘t’ (transpose) and ‘C’ or ‘c’ (conjugate transpose) that are often used as parameters to legacy BLAS implementations.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUBLAS_OP_N</td>
<td>the non-transpose operation is selected</td>
</tr>
<tr>
<td>CUBLAS_OP_T</td>
<td>the transpose operation is selected</td>
</tr>
<tr>
<td>CUBLAS_OP_C</td>
<td>the conjugate transpose operation is selected</td>
</tr>
</tbody>
</table>

2.2.1.4. **cusolverEigType_t**

The `cusolverEigType_t` type indicates which type of eigenvalue solver is. Its values correspond to Fortran integer 1 \(Ax = \lambda Bx\), 2 \(ABx = \lambda x\), 3 \(BAx = \lambda x\), used as parameters to legacy LAPACK implementations.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_EIG_TYPE_1</td>
<td>(Ax = \lambda Bx)</td>
</tr>
<tr>
<td>CUSOLVER_EIG_TYPE_2</td>
<td>(ABx = \lambda x)</td>
</tr>
<tr>
<td>CUSOLVER_EIG_TYPE_3</td>
<td>(BAx = \lambda x)</td>
</tr>
</tbody>
</table>

2.2.1.5. **cusolverEigMode_t**

The `cusolverEigMode_t` type indicates whether or not eigenvectors are computed. Its values correspond to Fortran character ‘N’ (only eigenvalues are computed), ‘V’ (both eigenvalues and eigenvectors are computed) used as parameters to legacy LAPACK implementations.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
CU-SOLVER Library

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_EIG_MODE_NOVECTOR</td>
<td>only eigenvalues are computed</td>
</tr>
<tr>
<td>CUSOLVER_EIG_MODE_VECTOR</td>
<td>both eigenvalues and eigenvectors are computed</td>
</tr>
</tbody>
</table>

2.2.1.6. cusolverIRSRefinement_t

The cusolverIRSRefinement_t type indicates which solver type would be used for the specific cusolver function. Most of our experimentation shows that CUSOLVER_IRS_REFINE_GMRES is the best option.


<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER IRS_REFINE_NOT_SET</td>
<td>Solver is not set, this value is what is set when creating the params structure. IRS solver will return error.</td>
</tr>
<tr>
<td>CUSOLVER IRS_REFINE_NONE</td>
<td>No refinement solver, the IRS solver performs a factorisation followed by a solve without any refinement. For example if the IRS solver was cusolverDnIRSXgesv(), this is equivalent to a Xgesv routine without refinement and where the factorisation is carried out in the lowest precision. If for example the main precision was CUSOLVER_R_64F and the lowest was CUSOLVER_R_64F as well, then this is equivalent to a call to cusolverDnDgesv().</td>
</tr>
<tr>
<td>CUSOLVER IRS_REFINE_CLASSICAL</td>
<td>Classical iterative refinement solver. Similar to the one used in LAPACK routines.</td>
</tr>
<tr>
<td>CUSOLVER IRS_REFINE_GMRES</td>
<td>GMRES (Generalized Minimal Residual) based iterative refinement solver. In recent study, the GMRES method has drawn the scientific community attention for its ability to be used as refinement solver that outperforms the classical iterative refinement method. based on our experimentation, we recommend this setting.</td>
</tr>
<tr>
<td>CUSOLVER IRS_REFINE_CLASSICAL_GMRES</td>
<td>Classical iterative refinement solver that uses the GMRES (Generalized Minimal Residual) internally to solve the correction equation at each iteration. We call the classical refinement iteration the outer iteration while the GMRES is called inner iteration. Note that if the tolerance of the inner GMRES is set very low, let say to machine precision, then the outer classical refinement iteration will performs only one iteration and thus this option will behaves like CUSOLVER IRS_REFINE_GMRES.</td>
</tr>
<tr>
<td>CUSOLVER IRS_REFINE_GMRES_GMRES</td>
<td>Similar to CUSOLVER IRS_REFINE_CLASSICAL_GMRES</td>
</tr>
</tbody>
</table>
2.2.1.7. cusolverDnIRSParams_t

This is a pointer type to an opaque cusolverDnIRSParams_t structure, which holds parameters for the iterative refinement linear solvers such as cusolverDnXgesv(). Use corresponding helper functions described below to either Create/Destroy this structure or Set/Get solver parameters.

2.2.1.8. cusolverDnIRSInfos_t

This is a pointer type to an opaque cusolverDnIRSInfos_t structure, which holds information about the performed call to an iterative refinement linear solver [e.g., cusolverDnXgesv()]. Use corresponding helper functions described below to either Create/Destroy this structure or retrieve solve information.

2.2.1.9. cusolverDnFunction_t

The cusolverDnFunction_t type indicates which routine needs to be configured by cusolverDnSetAdvOptions(). The value CUSOLVERDN_GETRF corresponds to the routine Getrf.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVERDN_GETRF</td>
<td>corresponds to Getrf</td>
</tr>
</tbody>
</table>

2.2.1.10. cusolverAlgMode_t

The cusolverAlgMode_t type indicates which algorithm is selected by cusolverDnSetAdvOptions(). The set of algorithms supported for each routine is described in detail along with the routine documentation.

The default algorithm is CUSOLVER_ALG_0. The user can also provide NULL to use the default algorithm.

2.2.1.11. cusolverStatus_t

This is the same as cusolverStatus_t in the sparse LAPACK section.

2.2.2. cuSolverSP Types

The float, double, cuComplex, and cuDoubleComplex data types are supported. The first two are standard C data types, while the last two are exported from cuComplex.h.
2.2.2.1. cusolverSpHandle_t

This is a pointer type to an opaque cuSolverSP context, which the user must initialize by calling cusolverSpCreate() prior to calling any other library function. An un-initialized Handle object will lead to unexpected behavior, including crashes of cuSolverSP. The handle created and returned by cusolverSpCreate() must be passed to every cuSolverSP function.

2.2.2.2. cusparseMatDescr_t

We have chosen to keep the same structure as exists in cuSparse to describe the shape and properties of a matrix. This enables calls to either cuSparse or cuSolver using the same matrix description.

```
typedef struct {
    cusparseMatrixType_t MatrixType;
    cusparseFillMode_t FillMode;
    cusparseDiagType_t DiagType;
    cusparseIndexBase_t IndexBase;
} cusparseMatDescr_t;
```

Please read documentation of CUSPARSE Library to understand each field of cusparseMatDescr_t.

2.2.2.3. cusolverStatus_t

This is a status type returned by the library functions and it can have the following values.

<table>
<thead>
<tr>
<th>cudaeStatus_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>The cuSolver library was not initialized. This is usually caused by the lack of a prior call, an error in the CUDA Runtime API called by the cuSolver routine, or an error in the hardware setup.</td>
</tr>
<tr>
<td></td>
<td><strong>To correct:</strong> call cusolverCreate() prior to the function call; and check that the hardware, an appropriate version of the driver, and the cuSolver library are correctly installed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>Resource allocation failed inside the cuSolver library. This is usually caused by a cudaMalloc() failure.</td>
</tr>
<tr>
<td></td>
<td><strong>To correct:</strong> prior to the function call, deallocate previously allocated memory as much as possible.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>An unsupported value or parameter was passed to the function [a negative vector size, for example].</td>
</tr>
<tr>
<td></td>
<td><strong>To correct:</strong> ensure that all the parameters being passed have valid values.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td></td>
</tr>
</tbody>
</table>
The function requires a feature absent from the device architecture; usually caused by the lack of support for atomic operations or double precision.

**To correct:** compile and run the application on a device with compute capability 2.0 or above.

**CUSOLVER_STATUS_EXECUTION_FAILED**

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.

**To correct:** check that the hardware, an appropriate version of the driver, and the cuSolver library are correctly installed.

**CUSOLVER_STATUS_INTERNAL_ERROR**

An internal cuSolver operation failed. This error is usually caused by a cudaMemcpyAsync() failure.

**To correct:** check that the hardware, an appropriate version of the driver, and the cuSolver library are correctly installed. Also, check that the memory passed as a parameter to the routine is not being deallocated prior to the routine’s completion.

**CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED**

The matrix type is not supported by this function. This is usually caused by passing an invalid matrix descriptor to the function.

**To correct:** check that the fields in descrA were set correctly.

### 2.2.3. cuSolverRF Types

cuSolverRF only supports double.

#### 2.2.3.1. cusolverRfHandle_t

The `cusolverRfHandle_t` is a pointer to an opaque data structure that contains the cuSolverRF library handle. The user must initialize the handle by calling `cusolverRfCreate()` prior to any other cuSolverRF library calls. The handle is passed to all other cuSolverRF library calls.

#### 2.2.3.2. cusolverRfMatrixFormat_t

The `cusolverRfMatrixFormat_t` is an enum that indicates the input/output matrix format assumed by the `cusolverRfSetupDevice()`, `cusolverRfSetupHost()`, `cusolverRfResetValues()`, `cusolveRfExtractBundledFactorsHost()` and `cusolverRfExtractSplitFactorsHost()` routines.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_MATRIX_FORMAT_CSR</td>
<td>matrix format CSR is assumed. [default]</td>
</tr>
<tr>
<td>CUSOLVER_MATRIX_FORMAT_CSC</td>
<td>matrix format CSC is assumed.</td>
</tr>
</tbody>
</table>
2.2.3.3.  `cusolverRfNumericBoostReport_t`

The `cusolverRfNumericBoostReport_t` is an enum that indicates whether numeric boosting (of the pivot) was used during the `cusolverRfRefactor()` and `cusolverRfSolve()` routines. The numeric boosting is disabled by default.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_NUMERIC_BOOST_NOT_USED</td>
<td>numeric boosting not used. (default)</td>
</tr>
<tr>
<td>CUSOLVER_NUMERIC_BOOST_USED</td>
<td>numeric boosting used.</td>
</tr>
</tbody>
</table>

2.2.3.4.  `cusolverRfResetValuesFastMode_t`

The `cusolverRfResetValuesFastMode_t` is an enum that indicates the mode used for the `cusolverRfResetValues()` routine. The fast mode requires extra memory and is recommended only if very fast calls to `cusolverRfResetValues()` are needed.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_RESET_VALUES_FAST_MODE_OFF</td>
<td>fast mode disabled. (default)</td>
</tr>
<tr>
<td>CUSOLVER_RESET_VALUES_FAST_MODE_ON</td>
<td>fast mode enabled.</td>
</tr>
</tbody>
</table>

2.2.3.5.  `cusolverRfFactorization_t`

The `cusolverRfFactorization_t` is an enum that indicates which (internal) algorithm is used for refactorization in the `cusolverRfRefactor()` routine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_FACTORIZATION_ALG0</td>
<td>algorithm 0. (default)</td>
</tr>
<tr>
<td>CUSOLVER_FACTORIZATION_ALG1</td>
<td>algorithm 1.</td>
</tr>
<tr>
<td>CUSOLVER_FACTORIZATION_ALG2</td>
<td>algorithm 2. Domino-based scheme.</td>
</tr>
</tbody>
</table>

2.2.3.6.  `cusolverRfTriangularSolve_t`

The `cusolverRfTriangularSolve_t` is an enum that indicates which (internal) algorithm is used for triangular solve in the `cusolverRfSolve()` routine.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_TRIANGULAR_SOLVE_ALG1</td>
<td>algorithm 1. (default)</td>
</tr>
<tr>
<td>CUSOLVER_TRIANGULAR_SOLVE_ALG2</td>
<td>algorithm 2. Domino-based scheme.</td>
</tr>
</tbody>
</table>

2.2.3.7.  `cusolverRfUnitDiagonal_t`

The `cusolverRfUnitDiagonal_t` is an enum that indicates whether and where the unit diagonal is stored in the input/output triangular factors in the `cusolverRfSetupDevice()`, `cusolverRfSetupHost()` and `cusolverRfExtractSplitFactorsHost()` routines.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>
2.2.3.8. cusolverStatus_t

The `cusolverStatus_t` is an enum that indicates success or failure of the cuSolverRF library call. It is returned by all the cuSolver library routines, and it uses the same enumerated values as the sparse and dense Lapack routines.

2.3. cuSolver Formats Reference

2.3.1. Index Base Format

The CSR or CSC format requires either zero-based or one-based index for a sparse matrix A. The GLU library supports only zero-based indexing. Otherwise, both one-based and zero-based indexing are supported in cuSolver.

2.3.2. Vector (Dense) Format

The vectors are assumed to be stored linearly in memory. For example, the vector

\[ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \]

is represented as

\[(x_1 \ x_2 \ \cdots \ x_n)\]

2.3.3. Matrix (Dense) Format

The dense matrices are assumed to be stored in column-major order in memory. The sub-matrix can be accessed using the leading dimension of the original matrix. For example, the m*n (sub-)matrix

\[
\begin{bmatrix}
        a_{1,1} & \cdots & a_{1,n} \\
        a_{2,1} & \cdots & a_{2,n} \\
         \vdots &       \ddots & \vdots \\
        a_{m,1} & \cdots & a_{m,n}
\end{bmatrix}
\]

is represented as
Using the CUSOLVER API

2.3.4. Matrix (CSR) Format

In CSR format the matrix is represented by the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Size</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>(int)</td>
<td>n</td>
<td>the number of rows [and columns] in the matrix.</td>
</tr>
<tr>
<td>nnz</td>
<td>(int)</td>
<td></td>
<td>the number of non-zero elements in the matrix.</td>
</tr>
<tr>
<td>csrRowPtr</td>
<td>(int *)</td>
<td>n+1</td>
<td>the array of offsets corresponding to the start of each row in the arrays csrColInd and csrVal. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix.</td>
</tr>
<tr>
<td>csrColInd</td>
<td>(int *)</td>
<td>nnz</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. <strong>It is assumed that this array is sorted by row and by column within each row.</strong></td>
</tr>
<tr>
<td>csrVal</td>
<td>(S</td>
<td>D</td>
<td>C</td>
</tr>
</tbody>
</table>

Note that in our CSR format sparse matrices are assumed to be stored in row-major order, in other words, the index arrays are first sorted by row indices and then within each row by column indices. Also it is assumed that each pair of row and column indices appears only once.

For example, the $4 \times 4$ matrix

$$
A = \begin{pmatrix}
1.0 & 3.0 & 0.0 & 0.0 \\
0.0 & 4.0 & 6.0 & 0.0 \\
2.0 & 5.0 & 7.0 & 8.0 \\
0.0 & 0.0 & 0.0 & 9.0 \\
\end{pmatrix}
$$

is represented as

$$
csrRowPtr = (0 \ 2 \ 4 \ 8 \ 9)
$$
2.3.5. Matrix (CSC) Format

In CSC format the matrix is represented by the following parameters

<table>
<thead>
<tr>
<th>parameter</th>
<th>type</th>
<th>size</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>(int)</td>
<td></td>
<td>the number of rows (and columns) in the matrix.</td>
</tr>
<tr>
<td>nnz</td>
<td>(int)</td>
<td></td>
<td>the number of non-zero elements in the matrix.</td>
</tr>
<tr>
<td>cscColPtr</td>
<td>(int *)</td>
<td>n+1</td>
<td>the array of offsets corresponding to the start of each column in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>arrays cscRowInd and cscVal. This array has also an extra entry at the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>end that stores the number of non-zero elements in the matrix.</td>
</tr>
<tr>
<td>cscRowInd</td>
<td>(int *)</td>
<td>nnz</td>
<td>the array of row indices corresponding to the non-zero elements in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>matrix. It is assumed that this array is sorted by column and by row</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>within each column.</td>
</tr>
<tr>
<td>cscVal</td>
<td>(S</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>It is assumed that this array is sorted by column and by row within each</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>column.</td>
</tr>
</tbody>
</table>

Note that in our CSC format sparse matrices are assumed to be stored in column-major order, in other words, the index arrays are first sorted by column indices and then within each column by row indices. Also it is assumed that each pair of row and column indices appears only once.

For example, the 4x4 matrix

\[
A = \begin{bmatrix}
1.0 & 3.0 & 0.0 & 0.0 \\
0.0 & 4.0 & 6.0 & 0.0 \\
2.0 & 5.0 & 7.0 & 8.0 \\
0.0 & 0.0 & 0.0 & 9.0 \\
\end{bmatrix}
\]

is represented as

\[
cscColPtr = (0 \ 2 \ 5 \ 7 \ 9)
\]

\[
cscRowInd = (0 \ 2 \ 0 \ 1 \ 2 \ 1 \ 2 \ 3)
\]

\[
cscVal = (1.0 \ 2.0 \ 3.0 \ 4.0 \ 5.0 \ 6.0 \ 7.0 \ 8.0 \ 9.0)
\]
2.4. cuSolverDN: dense LAPACK Function Reference

This chapter describes the API of cuSolverDN, which provides a subset of dense LAPACK functions.

2.4.1. cuSolverDN Helper Function Reference

The cuSolverDN helper functions are described in this section.

2.4.1.1. cusolverDnCreate()

```c
cusolverStatus_t cusolverDnCreate(cusolverDnHandle_t *handle);
```

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.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the pointer to the handle to the cuSolverDN context.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the initialization succeeded.
- CUSOLVER_STATUS_NOT_INITIALIZED: the CUDA Runtime initialization failed.
- CUSOLVER_STATUS_ALLOC_FAILED: the resources could not be allocated.
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.

2.4.1.2. cusolverDnDestroy()

```c
cusolverStatus_t cusolverDnDestroy(cusolverDnHandle_t handle);
```

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the shutdown succeeded.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
2.4.1.3. cusolverDnSetStream()

cusolverStatus_t
cusolverDnSetStream(cusolverDnHandle_t handle, cudaStream_t streamId)

This function sets the stream to be used by the cuSolverDN library to execute its routines.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>streamId</td>
<td>host</td>
<td>input</td>
<td>the stream to be used by the library.</td>
</tr>
</tbody>
</table>

Status Returned

- CUSOLVER_STATUS_SUCCESS: the stream was set successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.

2.4.1.4. cusolverDnGetStream()

cusolverStatus_t
cusolverDnGetStream(cusolverDnHandle_t handle, cudaStream_t *streamId)

This function sets the stream to be used by the cuSolverDN library to execute its routines.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>streamId</td>
<td>host</td>
<td>output</td>
<td>the stream to be used by the library.</td>
</tr>
</tbody>
</table>

Status Returned

- CUSOLVER_STATUS_SUCCESS: the stream was set successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.

2.4.1.5. cusolverDnCreateSyevjInfo()

cusolverStatus_t
cusolverDnCreateSyevjInfo(    syevjInfo_t *info);    

This function creates and initializes the structure of syevj, syevjBatched and sygvj to default values.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>the pointer to the structure of syevj.</td>
</tr>
</tbody>
</table>

Status Returned

- CUSOLVER_STATUS_SUCCESS: the structure was initialized successfully.
- CUSOLVER_STATUS_ALLOC_FAILED: the resources could not be allocated.
2.4.1.6. cusolverDnDestroySyevjInfo()

cusolverStatus_t
cusolverDnDestroySyevjInfo(
    syevjInfo_t info);

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>input</td>
<td>the structure of syevj.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS the resources are released successfully.

2.4.1.7. cusolverDnXsyevjSetTolerance()

cusolverStatus_t
cusolverDnXsyevjSetTolerance(
    syevjInfo_t info,
    double tolerance);

This function configures tolerance of syevj.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of syevj.</td>
</tr>
<tr>
<td>tolerance</td>
<td>host</td>
<td>input</td>
<td>accuracy of numerical eigenvalues.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS the operation completed successfully.

2.4.1.8. cusolverDnXsyevjSetMaxSweeps()

cusolverStatus_t
cusolverDnXsyevjSetMaxSweeps(
    syevjInfo_t info,
    int max_sweeps);

This function configures maximum number of sweeps in syevj. The default value is 100.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of syevj.</td>
</tr>
<tr>
<td>max_sweeps</td>
<td>host</td>
<td>input</td>
<td>maximum number of sweeps.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS the operation completed successfully.

2.4.1.9. cusolverDnXsyevjSetSortEig()

cusolverStatus_t
Using the CUSOLVER API

cusolverDnXsyevjSetSortEig(
    syevjInfo_t info,
    int sort_eig)

if sort_eig is zero, the eigenvalues are not sorted. This function only works for syevjBatched. syevj and sygvj always sort eigenvalues in ascending order. By default, eigenvalues are always sorted in ascending order.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of syevj.</td>
</tr>
<tr>
<td>sort_eig</td>
<td>host</td>
<td>input</td>
<td>if sort_eig is zero, the eigenvalues are not sorted.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS: the operation completed successfully.

2.4.1.10. cusolverDnXsyevjGetResidual()

cusolverStatus_t
cusolverDnXsyevjGetResidual(
    cusolverDnHandle_t handle,
    syevjInfo_t info,
    double *residual)

This function reports residual of syevj or sygvj. It does not support syevjBatched. If the user calls this function after syevjBatched, the error CUSOLVER_STATUS_NOT_SUPPORTED is returned.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>input</td>
<td>the pointer to the structure of syevj.</td>
</tr>
<tr>
<td>residual</td>
<td>host</td>
<td>output</td>
<td>residual of syevj.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
CUSOLVER_STATUS_NOT_SUPPORTED: does not support batched version

2.4.1.11. cusolverDnXsyevjGetSweeps()

cusolverStatus_t
cusolverDnXsyevjGetSweeps(
    cusolverDnHandle_t handle,
    syevjInfo_t info,
    int *executed_sweeps)

This function reports number of executed sweeps of syevj or sygvj. It does not support syevjBatched. If the user calls this function after syevjBatched, the error CUSOLVER_STATUS_NOT_SUPPORTED is returned.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

### DU-06709-001_v11.1   |   21

<table>
<thead>
<tr>
<th>info</th>
<th>host</th>
<th>input</th>
<th>the pointer to the structure of syevj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>executed_sweeps</td>
<td>host</td>
<td>output</td>
<td>number of executed sweeps.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_SUPPORTED: does not support batched version

#### 2.4.1.12. `cusolverDnCreateGesvdjInfo()`

```c
cusolverStatus_t
cusolverDnCreateGesvdjInfo(
    gesvdjInfo_t *info);
```

This function creates and initializes the structure of gesvdj and gesvdjBatched to default values.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>the pointer to the structure of gesvdj.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the structure was initialized successfully.
- CUSOLVER_STATUS_ALLOC_FAILED: the resources could not be allocated.

#### 2.4.1.13. `cusolverDnDestroyGesvdjInfo()`

```c
cusolverStatus_t
cusolverDnDestroyGesvdjInfo(
    gesvdjInfo_t info);
```

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>input</td>
<td>the structure of gesvdj.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the resources are released successfully.

#### 2.4.1.14. `cusolverDnXgesvdjSetTolerance()`

```c
cusolverStatus_t
cusolverDnXgesvdjSetTolerance(
    gesvdjInfo_t info,
    double tolerance)
```

This function configures tolerance of gesvdj.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of gesvdj.</td>
</tr>
<tr>
<td>tolerance</td>
<td>host</td>
<td>input</td>
<td>accuracy of numerical singular values.</td>
</tr>
</tbody>
</table>
2.4.1.15. `cusolverDnXgesvdjSetMaxSweeps()`

```c
cusolverStatus_t
cusolverDnXgesvdjSetMaxSweeps(
    gesvdjInfo_t info,
    int max_sweeps)
```

This function configures maximum number of sweeps in `gesvdj`. The default value is 100.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of <code>gesvdj</code>.</td>
</tr>
<tr>
<td>max_sweeps</td>
<td>host</td>
<td>input</td>
<td>maximum number of sweeps.</td>
</tr>
</tbody>
</table>

Status Returned

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully. |

2.4.1.16. `cusolverDnXgesvdjSetSortEig()`

```c
cusolverStatus_t
cusolverDnXgesvdjSetSortEig(
    gesvdjInfo_t info,
    int sort_svd)
```

If `sort_svd` is zero, the singular values are not sorted. This function only works for `gesvdjBatched`. `gesvdj` always sorts singular values in descending order. By default, singular values are always sorted in descending order.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of <code>gesvdj</code>.</td>
</tr>
<tr>
<td>sort_svd</td>
<td>host</td>
<td>input</td>
<td>if <code>sort_svd</code> is zero, the singular values are not sorted.</td>
</tr>
</tbody>
</table>

Status Returned

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully. |

2.4.1.17. `cusolverDnXgesvdjGetResidual()`

```c
cusolverStatus_t
cusolverDnXgesvdjGetResidual(
    cusolverDnHandle_t handle,
    gesvdjInfo_t info,
    double *residual)
```

This function reports residual of `gesvdj`. It does not support `gesvdjBatched`. If the user calls this function after `gesvdjBatched`, the error `CUSOLVER_STATUS_NOT_SUPPORTED` is returned.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
</table>

```
Using the CUSOLVER API

2.4.1.18. cusolverDnXgesvdjGetSweeps()

cusolverStatus_t
cusolverDnXgesvdjGetSweeps(
    cusolverDnHandle_t handle,  
    gesvdjInfo_t info,  
    int *executed_sweeps)

This function reports number of executed sweeps of gesvdj. It does not support gesvdjBatched. If the user calls this function after gesvdjBatched, the error CUSOLVER_STATUS_NOT_SUPPORTED is returned.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>input</td>
<td>the pointer to the structure of gesvdj.</td>
</tr>
<tr>
<td>executed_sweeps</td>
<td>host</td>
<td>output</td>
<td>number of executed sweeps.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_SUPPORTED</td>
<td>does not support batched version</td>
</tr>
</tbody>
</table>

2.4.1.19. cusolverDnIRSParamsCreate()

cusolverStatus_t
cusolverDnIRSParamsCreate(cusolverDnIRSParams_t *params);

This function creates and initializes the structure of parameters for an IRS solver such as the cusolverDnIRSXgesv() or the cusolverDnIRSXgels() functions to default values. The params structure created by this function can be used by one or more call to the same or to a different IRS solver. Note that, in CUDA-10.2, the behavior was different and a new params structure was needed to be created per each call to an IRS solver. Also note that, the user can also change configurations of the params then call a new IRS instance, but be careful that the previous call was done because any change to the configuration before that the previous call was done could affect it.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>output</td>
<td>Pointer to the cusolverDnIRSParams_t Params structure</td>
</tr>
</tbody>
</table>

Status Returned
Using the CUSOLVER API

CUSOLVER_STATUS_SUCCESS | The structure was created and initialized successfully.
CUSOLVER_STATUS_ALLOC_FAILED | The resources could not be allocated.

### 2.4.1.20. cusolverDnIRSParamsDestroy()

```c
cusolverStatus_t
cusolverDnIRSParamsDestroy(cusolverDnIRSParams_t params);
```

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>The resources are released successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_INFOS_NOT_DESTROYED</td>
<td>Not all the Infos structure associated with this Params structure have been destroyed yet.</td>
</tr>
</tbody>
</table>

### 2.4.1.21. cusolverDnIRSParamsSetSolverPrecisions()

```c
cusolverStatus_t
cusolverDnIRSParamsSetSolverPrecisions(
    cusolverDnIRSParams_t params,
    cusolverPrecType_t solver_main_precision,
    cusolverPrecType_t solver_lowest_precision);
```

This function sets both, the main and the lowest precision for the Iterative Refinement Solver (IRS). By main precision, we mean the precision of the Input and Output datatype. By lowest precision, we mean the solver is allowed to use as lowest computational precision during the LU factorization process. Note that, the user has to set both the main and lowest precision before a first call to the IRS solver because they are NOT set by default with the params structure creation, as it depends on the Input Output data type and user request. It is a wrapper to both cusolverDnIRSParamsSetSolverMainPrecision() and cusolverDnIRSParamsSetSolverLowestPrecision(). All possible combinations of main/lowest precision are described in the table below. Usually the lowest precision defines the speedup that can be achieved. The ratio of the performance of the lowest precision over the main precision (e.g., Inputs/Outputs datatype) define somehow the upper bound of the speedup that could be obtained. More precisely, it depends on many factors, but for large matrices sizes, it is the ratio of the matrix-matrix rank-k product (e.g., GEMM where K is 256 and M=N=size of the matrix) that define the possible speedup. For instance, if the inout precision is real double precision CUSOLVER_R_64F and the lowest precision is CUSOLVER_R_32F, then we can expect a speedup of at most 2X for large problem sizes. If the lowest precision was CUSOLVER_R_16F, then we can expect 3X-4X. A reasonable strategy should take the number of right-hand sides, the size of the matrix as well as the convergence rate into account.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
</table>

cuSOLVER Library
Using the CUSOLVER API

The cusolverDnIRSParams_t Params structure

<table>
<thead>
<tr>
<th>params</th>
<th>host</th>
<th>in/out</th>
<th>The cusolverDnIRSParams_t Params structure</th>
</tr>
</thead>
</table>

solver_main_precision

<table>
<thead>
<tr>
<th>solver_main_precision</th>
<th>host</th>
<th>input</th>
<th>Allowed Inputs/Outputs datatype (for example CUSOLVER_R_FP64 for a real double precision data). See the table below for the supported precisions.</th>
</tr>
</thead>
</table>

solver_lowest_precision

<table>
<thead>
<tr>
<th>solver_lowest_precision</th>
<th>host</th>
<th>input</th>
<th>Allowed lowest compute type (for example CUSOLVER_R_16F for half precision computation). See the table below for the supported precisions.</th>
</tr>
</thead>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>The operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
</tbody>
</table>

Supported Inputs/Outputs data type and lower precision for the IRS solver

<table>
<thead>
<tr>
<th>Inputs/Outputs Data Type (e.g., main precision)</th>
<th>Supported values for the lowest precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_C_64F</td>
<td>CUSOLVER_C_64F, CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32</td>
</tr>
<tr>
<td>CUSOLVER_C_32F</td>
<td>CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32</td>
</tr>
<tr>
<td>CUSOLVER_R_64F</td>
<td>CUSOLVER_R_64F, CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32</td>
</tr>
<tr>
<td>CUSOLVER_R_32F</td>
<td>CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32</td>
</tr>
</tbody>
</table>

2.4.1.22. cusolverDnIRSParamsSetSolverMainPrecision()

cusolverStatus_t

cusolverDnIRSParamsSetSolverMainPrecision(
    cusolverDnIRSParams_t_params,
    cusolverPrecType_t solver_main_precision);

This function sets the main precision for the Iterative Refinement Solver (IRS). By main precision, we mean, the type of the Input and Output data. Note that, the user has to set both the main and lowest precision before a first call to the IRS solver because they are NOT set by default with the params structure creation, as it depends on the Input Output data type and user request. user can set it by either calling this function or by calling cusolverDnIRSParamsSetSolverPrecisions() which set both the main and the lowest precision together. All possible combinations of main/lowest precision are described in the table in the cusolverDnIRSParamsSetSolverPrecisions() section above.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
<tr>
<td>solver_main_precision</td>
<td>host</td>
<td>input</td>
<td>Allowed Inputs/Outputs datatype (for example CUSOLVER_R_FP64</td>
</tr>
</tbody>
</table>
for a real double precision data. See the table in the cusolverDnIRSParsSetSolverPrecisions() section above for the supported precisions.

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
</tbody>
</table>

#### 2.4.1.23. cusolverDnIRSParsSetSolverLowestPrecision()

```c

cusolverStatus_t
cusolverDnIRSParsSetSolverLowestPrecision(  
cusolverDnIRSPars_t params,  
cusolverPrecType_t lowest_precision_type);
```

This function sets the lowest precision that will be used by Iterative Refinement Solver. By lowest precision, we mean the solver is allowed to use as lowest computational precision during the LU factorization process. Note that, the user has to set both the main and lowest precision before a first call to the IRS solver because they are NOT set by default with the params structure creation, as it depends on the Input Output data type and user request. Usually the lowest precision defines the speedup that can be achieved. The ratio of the performance of the lowest precision over the main precision (e.g., Inputs/Outputs datatype define somehow the upper bound of the speedup that could be obtained. More precisely, it depends on many factors, but for large matrices sizes, it is the ratio of the matrix-matrix rank-k product (e.g., GEMM where K is 256 and M=N=size of the matrix) that define the possible speedup. For instance, if the inout precision is real double precision CUSOLVER_R_64F and the lowest precision is CUSOLVER_R_32F, then we can expect a speedup of at most 2X for large problem sizes. If the lowest precision was CUSOLVER_R_16F, then we can expect 3X-4X. A reasonable strategy should take the number of right-hand sides, the size of the matrix as well as the convergence rate into account.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSPars_t Params structure</td>
</tr>
<tr>
<td>lowest_precision_type</td>
<td>host</td>
<td>input</td>
<td>Allowed lowest compute type (for example CUSOLVER_R_16F for half precision computation). See the table in the cusolverDnIRSParsSetSolverPrecisions() section above for the supported precisions.</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
</tbody>
</table>
2.4.1.24. **cusolverDnIRSParamsSetRefinementSolver()**

cusolverStatus_t
cusolverDnIRSParamsSetRefinementSolver(
   cusolverDnIRSParams_t params,
   cusolverIRSRefinement_t solver);

This function sets the refinement solver to be used in the Iterative Refinement Solver functions such as the `cusolverDnIRSXgesv()` or the `cusolverDnIRSXgels()` functions. Note that, the user has to set the refinement algorithm before a first call to the IRS solver because it is NOT set by default with the creating of params. Details about values that can be set to and theirs meaning are described in the table below.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
<tr>
<td>solver</td>
<td>host</td>
<td>input</td>
<td>Type of the refinement solver to be used by the IRS solver such as cusolverDnIRSXgesv() or cusolverDnIRSXgels</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_IRS_REFINE_NOT_SET</td>
<td>Solver is not set, this value is what is set when creating the params structure. IRS solver will return error.</td>
</tr>
<tr>
<td>CUSOLVER_IRS_REFINE_NONE</td>
<td>No refinement solver, the IRS solver performs a factorization followed by a solve without any refinement. For example, if the IRS solver was cusolverDnIRSXgesv(), this is equivalent to a Xgesv routine without refinement and where the factorization is carried out in the lowest precision. If for example the main precision was CUSOLVER_R_64F and the lowest was CUSOLVER_R_64F as well, then this is equivalent to a call to cusolverDnDgesv().</td>
</tr>
<tr>
<td>CUSOLVER_IRS_REFINE_CLASSICAL</td>
<td>Classical iterative refinement solver. Similar to the one used in LAPACK routines.</td>
</tr>
<tr>
<td>CUSOLVER_IRS_REFINE_GMRES</td>
<td>GMRES (Generalized Minimal Residual) based iterative refinement solver. In recent study, the GMRES method has drawn the scientific community attention for its ability to be used as refinement solver that outperforms the classical iterative refinement method. Based on our experimentation, we recommend this setting.</td>
</tr>
</tbody>
</table>
CUSOLVER_IRS_REFINE_CLASSICAL_GMRES

CUSOLVER_IRS_REFINE_GMRES

CUSOLVER_IRS_REFINE_GMRES_GMRES

2.4.1.25. cusolverDnIRSParamsSetTol()

cusolverStatus_t
cusolverDnIRSParamsSetTol( 
    cusolverDnIRSParams_t params,
    double val );

This function sets the tolerance for the refinement solver. By default it is such that, all the RHS satisfy:

\[ \text{RNRM} < \sqrt{N} \times \text{XNRM} \times \text{ANRM} \times \text{EPS} \times \text{BWDMAX} \]

where

- RNRM is the infinity-norm of the residual
- XNRM is the infinity-norm of the solution
- ANRM is the infinity-operator-norm of the matrix A
- EPS is the machine epsilon for the Inputs/Outputs datatype that matches LAPACK `<X>LAMCH(\text{\textquoteleft Epsilon\textquoteright})`
- BWDMAX, the value BWDMAX is fixed to 1.0

The user can use this function to change the tolerance to a lower or higher value. Our goal is to give the user more control such a way he can investigate and control every detail of the IRS solver. Note the, the tolerance value is always in real double precision whatever the Inputs/Outputs datatype is.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
<tr>
<td>val</td>
<td>host</td>
<td>input</td>
<td>double precision real value to which the refinement tolerance will be set</td>
</tr>
</tbody>
</table>

Status Returned
CUSOLVER_STATUS_SUCCESS | The operation completed successfully.
CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED | The Params structure was not created.

2.4.1.26. cusolverDnIRSParamsSetTolInner()

cusolverStatus_t
cusolverDnIRSParamsSetTolInner(
    cusolverDnIRSParams_t params,
    double val);

This function sets the tolerance for the inner refinement solver when the refinement solver consists of two-levels solver [e.g., CUSOLVER_IRS_REFINE_CLASSICAL_GMRES or CUSOLVER_IRS_REFINE_GMRES_GMRES cases]. It is not referenced in case of one level refinement solver such as CUSOLVER_IRS_REFINE_CLASSICAL or CUSOLVER_IRS_REFINE_GMRES. It is set to 1e-4 by default. This function set the tolerance for the inner solver [e.g. the inner GMRES]. For example, if the Refinement Solver was set to CUSOLVER_IRS_REFINE_CLASSICAL_GMRES, setting this tolerance mean that the inner GMRES solver will converge to that tolerance at each outer iteration of the classical refinement solver. Our goal is to give the user more control such a way he can investigate and control every detail of the IRS solver. Note the, the tolerance value is always in real double precision whatever the Inputs/Outputs datatype is.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
<tr>
<td>val</td>
<td>host</td>
<td>input</td>
<td>Double precision real value to which the tolerance of the inner refinement solver will be set.</td>
</tr>
</tbody>
</table>

Status Returned

CUSOLVER_STATUS_SUCCESS | The operation completed successfully.
CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED | The Params structure was not created.

2.4.1.27. cusolverDnIRSParamsSetMaxIters()

cusolverStatus_t
cusolverDnIRSParamsSetMaxIters(
    cusolverDnIRSParams_t params,
    int max_iters);

This function sets the total number of allowed refinement iterations after which the solver will stop. Total means any iteration which means the sum of the outer and the inner iterations (inner is meaningful when two-levels refinement solver is set). Default value is set to 50. Our goal is to give the user more control such a way he can investigate and control every detail of the IRS solver.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
</tbody>
</table>
2.4.1.28. cusolverDnIRSParamsSetMaxItersInner()

```c
 cusolverStatus_t
 cusolverDnIRSParamsSetMaxItersInner(    cusolverDnIRSParams_t params,
     cusolver_int_t maxiters_inner);
```

This function sets the maximal number of iterations allowed for the inner refinement solver. It is not referenced in case of one level refinement solver such as CUSOLVER_IRS_REFINE_CLASSICAL or CUSOLVER_IRS_REFINE_GMRES. The inner refinement solver will stop after reaching either the inner tolerance or the MaxItersInner value. By default, it is set to 50. Note that this value could not be larger than the MaxIters since MaxIters is the total number of allowed iterations. Note that, if the user call cusolverDnIRSParamsSetMaxIters after calling this function, the SetMaxIters has priority and will overwrite MaxItersInner to the minimum value of (MaxIters, MaxItersInner).

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
<tr>
<td>maxiters_inner</td>
<td>host</td>
<td>input</td>
<td>Maximum number of allowed inner iterations for the inner refinement solver. Meaningful when the refinement solver is a two-levels solver such as CUSOLVER_IRS_REFINE_CLASSICAL_GMRES or CUSOLVER_IRS_REFINE_GMRES_GMRES. Value should be less or equal to MaxIters.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>The operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_INVALID</td>
<td>if the value was larger than MaxIters.</td>
</tr>
</tbody>
</table>

2.4.1.29. cusolverDnIRSParamsEnableFallback()

```c
 cusolverStatus_t
 cusolverDnIRSParamsEnableFallback(    cusolverDnIRSParams_t params );
```
This function enable the fallback to the main precision in case the Iterative Refinement Solver (IRS) failed to converge. In other term, if the IRS solver failed to converge, the solver will return a no convergence code (e.g., niter < 0), but can either return the non-convergent solution as it is (e.g., disable fallback) or can fallback (e.g., enable fallback) to the main precision (which is the precision of the Inputs/Outputs data) and solve the problem from scratch returning the good solution. This is the behavior by default, and it will guarantee that the IRS solver always provide the good solution. This function is provided because we provided cusolverDnIRSParamsDisableFallback which allow the user to disable the fallback and thus this function allow him to re-enable it.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: The operation completed successfully.
- CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED: The Params structure was not created.

### 2.4.1.30. cusolverDnIRSParamsDisableFallback()

```c
CUSOLVER_STATUS_SUCCESS

cusolverDnIRSParamsDisableFallback(
    cusolverDnIRSParams_t params);
```

This function disables the fallback to the main precision in case the Iterative Refinement Solver (IRS) failed to converge. In other term, if the IRS solver failed to converge, the solver will return a no convergence code (e.g., niter < 0), but can either return the non-convergent solution as it is (e.g., disable fallback) or can fallback (e.g., enable fallback) to the main precision (which is the precision of the Inputs/Outputs data) and solve the problem from scratch returning the good solution. This function disables the fallback and the returned solution is whatever the refinement solver was able to reach before it returns. Disabling fallback does not guarantee that the solution is the good one. However, for some users, who want to keep getting the solution of the lower precision, in case the IRS did not converge after certain number of iterations, they need to disable the fallback. The user can re-enable it by calling cusolverDnIRSParamsEnableFallback.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSParams_t Params structure</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: The operation completed successfully.
- CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED: The Params structure was not created.

### 2.4.1.31. cusolverDnIRSParamsGetMaxIters()

```c
CUSOLVER_STATUS_SUCCESS

cusolverStatus_t cuSOLVER Library
```

DU-06709-001_v11.1 | 31
The function `cusolverDnIRSParamsGetMaxIters` returns the current setting in the params structure for the maximal allowed number of iterations (e.g., either the default MaxIters, or the one set by the user in case he set it using `cusolverDnIRSParamsSetMaxIters`). Note that, this function returns the current setting in the params configuration and not to be confused with the `cusolverDnIRSInfosGetMaxIters` which return the maximal allowed number of iterations for a particular call to an IRS solver. To be clearer, the params structure can be used for many calls to an IRS solver. A user can change the allowed MaxIters between calls while the Infos structure in `cusolverDnIRSInfosGetMaxIters` contains information about a particular call and cannot be reused for different call, and thus, `cusolverDnIRSInfosGetMaxIters` return the allowed MaxIters for that call.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>in</td>
<td>The <code>cusolverDnIRSParams_t</code> Params structure</td>
</tr>
<tr>
<td>maxiters</td>
<td>host</td>
<td>output</td>
<td>The maximal number of iterations that is currently set</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The Params structure was not created.</td>
</tr>
</tbody>
</table>

### 2.4.1.32. `cusolverDnIRSInfosCreate()`

The function `cusolverDnIRSInfosCreate` creates and initializes the Infos structure that will hold the refinement information of an Iterative Refinement Solver (IRS) call. Such information includes the total number of iterations that was needed to converge (`Niters`), the outer number of iterations (meaningful when two-levels preconditioner such as `CUSOLVER_IRS_REFINE_CLASSICAL_GMRES` is used), the maximal number of iterations that was allowed for that call, and a pointer to the matrix of the convergence history residual norms. The Infos structure need to be created before a call to an IRS solver. The Infos structure is valid for only one call to an IRS solver, since it holds info about that solve and thus each solve will requires its own Infos structure.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>Pointer to the <code>cusolverDnIRSInfos_t</code> Infos structure</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The structure was initialized successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>The resources could not be allocated.</td>
</tr>
</tbody>
</table>
2.4.1.33. cuSolverDnIRSInfosDestroy()

cusolverStatus_t
cusolverDnIRSInfosDestroy(
    cusolverDnIRSInfos_t infos);

This function destroys and releases any memory required by the Infos structure. This function destroys all the informations (e.g., Nitors performed, OuterNitors performed, residual history etc) about a solver call, thus, a user is supposed to call it once he is done from the informations he need.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>host</td>
<td>in/out</td>
<td>The cusolverDnIRSInfos_t Infos structure</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
</tr>
<tr>
<td>the resources are released successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED</td>
</tr>
<tr>
<td>The Infos structure was not created.</td>
</tr>
</tbody>
</table>

2.4.1.34. cuSolverDnIRSInfosGetMaxIters()

cusolverStatus_t
cusolverDnIRSInfosGetMaxIters(
    cusolverDnIRSInfos_t infos,
    cusolver_int_t *maxiters);

This function returns the maximal allowed number of iterations that was set for the corresponding call to the IRS solver. Note that, this function returns the setting that was set when that call happened and not to be confused with the cuSolverDnIRSParamsGetMaxIters which return the current setting in the params configuration structure. To be clearer, the params structure can be used for many calls to an IRS solver. A user can change the allowed MaxIters between calls while the Infos structure in cuSolverDnIRSInfosGetMaxIters contains information about a particular call and cannot be reused for different call, and thus cuSolverDnIRSInfosGetMaxIters return the allowed MaxIters for that call.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>infos</td>
<td>host</td>
<td>in</td>
<td>The cusolverDnIRSInfos_t Infos structure</td>
</tr>
<tr>
<td>maxiters</td>
<td>host</td>
<td>output</td>
<td>The maximal number of iterations that is currently set</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
</tr>
<tr>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED</td>
</tr>
<tr>
<td>The Infos structure was not created.</td>
</tr>
</tbody>
</table>

2.4.1.35. cuSolverDnIRSInfosGetNiters()

cusolverStatus_t
cusolverDnIRSInfosGetNiters(
    cusolverDnIRSInfos_t infos,
    cusolver_int_t *nIters);
This function returns the total number of iterations performed by the IRS solver. If it was negative it means that the IRS solver did not converge and if the user did not disable the fallback to full precision, then the fallback to a full precision solution happened and solution is good. Please refer to the description of negative niters values in the corresponding IRS linear solver functions such as `cusolverDnXgesv()` or `cusolverDnXgels()`.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>infos</td>
<td>host</td>
<td>in</td>
<td>The cusolverDnIRSInfos_t Infos structure</td>
</tr>
<tr>
<td>niters</td>
<td>host</td>
<td>output</td>
<td>The total number of iterations performed by the IRS solver</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: The operation completed successfully.
- CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED: The Infos structure was not created.

### 2.4.1.36. cusolverDnIRSInfosGetOuterNiters()

```c
cusolverStatus_t cusolverDnIRSInfosGetOuterNiters( 
  cusolverDnIRSInfos_t infos, 
  cusolver_int_t *outer_niters );
```

This function returns the number of iterations performed by the outer refinement loop of the IRS solver. When the refinement solver consists of a one level solver such as CUSOLVER_IRS_REFINE_CLASSICAL or CUSOLVER_IRS_REFINE_GMRES, it is the same as Niters. When the refinement solver consists of a two-levels solver such as CUSOLVER_IRS_REFINE_CLASSICAL_GMRES or CUSOLVER_IRS_REFINE_GMRES_GMRES, it is the number of iterations of the outer loop. See description of cusolverIRSRefinementSolver_t section for more details.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>infos</td>
<td>host</td>
<td>in</td>
<td>The cusolverDnIRSInfos_t Infos structure</td>
</tr>
<tr>
<td>outer_niters</td>
<td>host</td>
<td>output</td>
<td>The number of iterations of the outer refinement loop of the IRS solver</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: The operation completed successfully.
- CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED: The Infos structure was not created.

### 2.4.1.37. cusolverDnIRSInfosRequestResidual()

```c
cusolverStatus_t cusolverDnIRSInfosRequestResidual( 
  cusolverDnIRSInfos_t infos );
```

This function, once called, tell the IRS solver to store the convergence history (residual norms) of the refinement phase in a matrix, that could be accessed via a pointer returned by the `cusolverDnIRSInfosGetResidualHistory()` function.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The cusolverDnIRSInfos_t Infos structure

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED</td>
<td>The Infos structure was not created.</td>
</tr>
</tbody>
</table>

2.4.1.38. cusolverDnIRSInfosGetResidualHistory()

cusolverStatus_t cusolverDnIRSInfosGetResidualHistory(
  cusolverDnIRSInfos_t infos,
  void **residual_history);

If the user called cusolverDnIRSInfosRequestResidual() before the call to the
IRS function, then the IRS solver will store the convergence history (residual norms) of
the refinement phase in a matrix, that could be accessed via a pointer returned by this
function. The datatype of the residual norms depends on the input and output data type.
If the Inputs/Outputs datatype is double precision real or complex (CUSOLVER_R_FP64
or CUSOLVER_C_FP64), this residual will be of type real double precision (FP64)
double, otherwise if the Inputs/Outputs datatype is single precision real or complex
(CUSOLVER_R_FP32 or CUSOLVER_C_FP32), this residual will be real single precision FP32
float.

The residual history matrix consists of two columns (even for the multiple right-hand side case
NRHS) of MaxIters+1 row, thus a matrix of size (MaxIters+1,2). Only the first OuterNiters+1
rows contains the residual norms the other (e.g., OuterNiters+2:Maxiters+1) are garbage. On
the first column, each row "i" specify the total number of iterations happened till this outer
iteration "i" and on the second columns the residual norm corresponding to this outer iteration
"i". Thus, the first row (e.g., outer iteration "0") consists of the initial residual (e.g., the residual
before the refinement loop start) then the consecutive rows are the residual obtained at each
outer iteration of the refinement loop. Note, it only consists of the history of the outer loop.

If the refinement solver was CUSOLVER_IRS_REFINE_CLASSICAL or
CUSOLVER_IRS_REFINE_GMRES, then OuterNiters=Niters (Niters is the total number
of iterations performed) and there is Niters+1 rows of norms that correspond to the Niters outer
iterations.

If the refinement solver was CUSOLVER_IRS_REFINE_CLASSICAL_GMRES or
CUSOLVER_IRS_REFINE_GMRES_GMRES, then OuterNiters <= Niters corresponds to the
outer iterations performed by the outer refinement loop. Thus, there is OuterNiters+1 residual
norms where row "i" correspond to the outer iteration "i" and the first column specify the total
number of iterations (outer and inner) that were performed till this step the second columns
correspond to the residual norm at this step.

For example, let say the user specify CUSOLVER_IRS_REFINE_CLASSICAL_GMRES as a
refinement solver and let say it needed 3 outer iterations to converge and 4,3,3 inner iterations
at each outer respectively. This consists of 10 total iterations. Row 0 correspond to the first
residual before the refinement start, so it has 0 in its first column. On row 1 which correspond
to the outer iteration 1, it will be shown 4 (4 is the total number of iterations that were
performed till now) on row 2, it will be 7 and on row 3 it will be 10.
As summary, let define ldh=Maxiters+1, the leading dimension of the residual matrix. then residual_history[i] shows the total number of iterations performed at the outer iteration “i” and residual_history[i+ldh] correspond to the norm of the residual at this outer iteration.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>infos</td>
<td>host</td>
<td>in</td>
<td>The cusolverDnIRSInfos_t Infos structure</td>
</tr>
<tr>
<td>residual_history</td>
<td>host</td>
<td>output</td>
<td>Returns a void pointer to the matrix of the convergence history residual norms. See the description above for the relation between the residual norm datatype and the inout datatype.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>The operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED</td>
<td>The Infos structure was not created.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>This function was called without calling cusolverDnIRSInfosRequestResidual() in advance.</td>
</tr>
</tbody>
</table>

### 2.4.1.39. cusolverDnCreateParams()

```c
cusolverStatus_t
cusolverDnCreateParams(
    cusolverDnParams_t *params);
```

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>output</td>
<td>the pointer to the structure of 64-bit API.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the structure was initialized successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
</tbody>
</table>

### 2.4.1.40. cusolverDnDestroyParams()

```c
cusolverStatus_t
cusolverDnDestroyParams(
    cusolverDnParams_t params);
```

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

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>the structure of 64-bit API.</td>
</tr>
</tbody>
</table>

**Status Returned**
2.4.1.41. `cusolverDnSetAdvOptions()`

```c
 cusolverStatus_t
 cusolverDnSetAdvOptions ( 
    cusolverDnParams_t params, 
    cusolverDnFunction_t function, 
    cusolverAlgMode_t algo   );
```

This function configures algorithm `algo` of function, a 64-bit API routine.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>params</code></td>
<td>host</td>
<td>in/out</td>
<td>the pointer to the structure of 64-bit API.</td>
</tr>
<tr>
<td><code>function</code></td>
<td>host</td>
<td>input</td>
<td>which routine to be configured.</td>
</tr>
<tr>
<td><code>algo</code></td>
<td>host</td>
<td>input</td>
<td>which algorithm to be configured.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>wrong combination of function and algo</td>
</tr>
</tbody>
</table>

2.4.2. Dense Linear Solver Reference (legacy)

This chapter describes linear solver API of cuSolverDN, including Cholesky factorization, LU with partial pivoting, QR factorization and Bunch-Kaufman (LDLT) factorization.

2.4.2.1. `cusolverDn<t>potrf()`

These helper functions calculate the necessary size of work buffers.

```c
 cusolverStatus_t
 cusolverDnSpotrf_bufferSize(cusolverDnHandle_t handle, 
                           cublasFillMode_t uplo, 
                           int n, 
                           float *A, 
                           int lda, 
                           int *Lwork );
```

```c
 cusolverStatus_t
 cusolverDnDpotrf_bufferSize(cusolveDnHandle_t handle, 
                           cublasFillMode_t uplo, 
                           int n, 
                           double *A, 
                           int lda, 
                           int *Lwork );
```

```c
 cusolverStatus_t
 cusolverDnCpotrf_bufferSize(cusolverDnHandle_t handle, 
                           cublasFillMode_t uplo, 
                           int n, 
                           cuComplex *A, 
                           int lda, 
```
Using the CUSOLVER API

```c
int *Lwork);

cusolverStatus_t
cusolverDnZpotrf_bufferSize(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
int n,
cuDoubleComplex *A,
int lda,
int *Lwork);

The S and D data types are real valued single and double precision, respectively.
```

```c
cusolverStatus_t
cusolverDnSpotrf(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
int n,
float *A,
int lda,
float *Workspace,
int Lwork,
int *devInfo);
```

```

cusolverStatus_t
cusolverDnDpotrf(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
int n,
double *A,
int lda,
double *Workspace,
int Lwork,
int *devInfo);
```

The C and Z data types are complex valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnCpotrf(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
int n,
cuComplex *A,
int lda,
cuComplex *Workspace,
int Lwork,
int *devInfo);
```

```

cusolverStatus_t
cusolverDnZpotrf(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
int n,
cuDoubleComplex *A,
int lda,
cuDoubleComplex *Workspace,
int Lwork,
int *devInfo);
```

This function computes the Cholesky factorization of a Hermitian positive-definite matrix.

A is a $n \times n$ Hermitian matrix, only lower or upper part is meaningful. The input parameter uplo indicates which part of the matrix is used. The function would leave other part untouched.

If input parameter uplo is CUBLAS_FILL_MODE_LOWER, only lower triangular part of A is processed, and replaced by lower triangular Cholesky factor L.
$A = L^* L^H$

If input parameter `uplo` is `CUBLAS_FILL_MODE_UPPER`, only upper triangular part of $A$ is processed, and replaced by upper triangular Cholesky factor $U$.

$A = U^H U$

The user has to provide working space which is pointed by input parameter `Workspace`. The input parameter `Lwork` is size of the working space, and it is returned by `potrf_bufferSize()`.

If Cholesky factorization failed, i.e. some leading minor of $A$ is not positive definite, or equivalently some diagonal elements of $L$ or $U$ is not a real number. The output parameter `devInfo` would indicate smallest leading minor of $A$ which is not positive definite.

If output parameter `devInfo` = $-i$ (less than zero), the $i$-th parameter is wrong (not counting handle).

**API of `potrf`**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td><code>uplo</code></td>
<td>host</td>
<td>input</td>
<td>indicates if matrix $A$ lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td><code>A</code></td>
<td>device</td>
<td>in/out</td>
<td>$&lt;$type$&gt;$ array of dimension $lda \times n$ with $lda \geq \max(1, n)$.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td><code>Workspace</code></td>
<td>device</td>
<td>in/out</td>
<td>working space, $&lt;$type$&gt;$ array of size $Lwork$.</td>
</tr>
<tr>
<td><code>Lwork</code></td>
<td>host</td>
<td>input</td>
<td>size of <code>Workspace</code>, returned by <code>potrf_bufferSize</code>.</td>
</tr>
<tr>
<td><code>devInfo</code></td>
<td>device</td>
<td>output</td>
<td>if <code>devInfo</code> = 0, the Cholesky factorization is successful. if <code>devInfo</code> = $-i$, the $i$-th parameter is wrong (not counting handle). if <code>devInfo</code> = $i$, the leading minor of order $i$ is not positive definite.</td>
</tr>
</tbody>
</table>

**Status Returned**

- `CUSOLVER_STATUS_SUCCESS` the operation completed successfully.
- `CUSOLVER_STATUS_NOT_INITIALIZED` the library was not initialized.
- `CUSOLVER_STATUS_INVALID_VALUE` invalid parameters were passed ($n < 0$ or $lda < \max(1, n)$).
- `CUSOLVER_STATUS_ARCH_MISMATCH` the device only supports compute capability 2.0 and above.
- `CUSOLVER_STATUS_INTERNAL_ERROR` an internal operation failed.
2.4.2.2. cusolverDnPotrf() [DEPRECATED]

[[DEPRECATED]] use cusolverDnXpotrf() instead. The routine will be removed in the next major release.

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
cusolverDnPotrf_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasFillMode_t uplo,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType computeType,
    size_t *workspaceInBytes )
```

The routine below computes the Cholesky factorization of a Hermitian positive-definite matrix using the generic API interface.

```c
 cusolverStatus_t
cusolverDnPotrf(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasFillMode_t uplo,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType computeType,
    void *pBuffer,
    size_t workspaceInBytes,
    int *info )
```

$A$ is a $n \times n$ Hermitian matrix, only lower or upper part is meaningful. The input parameter uplo indicates which part of the matrix is used. The function would leave other part untouched.

If input parameter uplo is CUBLAS_FILL_MODE_LOWER, only lower triangular part of $A$ is processed, and replaced by lower triangular Cholesky factor $L$.

$$A = L \cdot L^H$$

If input parameter uplo is CUBLAS_FILL_MODE_UPPER, only upper triangular part of $A$ is processed, and replaced by upper triangular Cholesky factor $U$.

$$A = U^H \cdot U$$

The user has to provide working space which is pointed by input parameter pBuffer. The input parameter workspaceInBytes is size in bytes of the working space, and it is returned by cusolverDnPotrf_bufferSize().
If Cholesky factorization failed, i.e. some leading minor of \( A \) is not positive definite, or equivalently some diagonal elements of \( L \) or \( U \) is not a real number. The output parameter \( \text{info} \) would indicate smallest leading minor of \( A \) which is not positive definite.

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

Currently, \texttt{cusolverDnPotrf} supports only the default algorithm.

Table of algorithms supported by \texttt{cusolverDnPotrf}

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for \texttt{cusolverDnPotrf_bufferSize} and \texttt{cusolverDnPotrf}:

API of \texttt{potrf}

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by \texttt{cusolverDnSetAdvOptions}.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix ( A ) lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{dataTypeA} )</td>
<td>host</td>
<td>in</td>
<td>data type of array ( A ).</td>
</tr>
<tr>
<td>( A )</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension ( \text{lda} \times n ) with ( \text{lda} ) is not less than ( \max(1,n) ).</td>
</tr>
<tr>
<td>( \text{lda} )</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>in/out</td>
<td>Working space. Array of type \text{void} of size ( \text{workspaceInBytes} ) bytes.</td>
</tr>
<tr>
<td>workspaceInBytes</td>
<td>host</td>
<td>input</td>
<td>size in bytes of ( \text{pBuffer} ), returned by \texttt{cusolverDnPotrf_bufferSize}.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if ( \text{info} = 0 ), the Cholesky factorization is successful. if ( \text{info} = -i ), the ( i )-th parameter is wrong (not counting handle). if ( \text{info} = i ), the leading minor of order ( i ) is not positive definite.</td>
</tr>
</tbody>
</table>

The generic API has two different types, \( \text{dataTypeA} \) is data type of the matrix \( A \), \( \text{computeType} \) is compute type of the operation. \texttt{cusolverDnPotrf} only supports the following four combinations.

Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>( \text{DataTypeA} )</th>
<th>( \text{ComputeType} )</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CUDA_R_32F}</td>
<td>\texttt{CUDA_R_32F}</td>
<td>\texttt{SPOTRF}</td>
</tr>
<tr>
<td>\texttt{CUDA_R_64F}</td>
<td>\texttt{CUDA_R_64F}</td>
<td>\texttt{DPOTRF}</td>
</tr>
<tr>
<td>\texttt{CUDA_C_32F}</td>
<td>\texttt{CUDA_C_32F}</td>
<td>\texttt{CPOTRF}</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

### Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [n&lt;0 or lda&lt;max(1,n)].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.2.3. `cusolverDn<t>potrs()`

```c
 cusolverStatus_t
cusolverDnSpotrs(cusolverDnHandle_t handle,
                  cublasFillMode_t uplo,
                  int n,
                  int nrhs,
                  const float *A,
                  int lda,
                  float *B,
                  int ldb,
                  int *devInfo);
```

```c
 cusolverStatus_t
cusolverDnDpotrs(cusolverDnHandle_t handle,
                 cublasFillMode_t uplo,
                 int n,
                 int nrhs,
                 const double *A,
                 int lda,
                 double *B,
                 int ldb,
                 int *devInfo);
```

```c
 cusolverStatus_t
cusolverDnCpotrs(cusolverDnHandle_t handle,
                 cublasFillMode_t uplo,
                 int n,
                 int nrhs,
                 const cuComplex *A,
                 int lda,
                 cuComplex *B,
                 int ldb,
                 int *devInfo);
```

```c
 cusolverStatus_t
cusolverDnZpotrs(cusolverDnHandle_t handle,
                 cublasFillMode_t uplo,
                 int n,
                 int nrhs,
                 const cuDoubleComplex *A,
                 int lda,
                 cuDoubleComplex *B,
                 int ldb,
                 int *devInfo);
```
This function solves a system of linear equations

\[ A^*X = B \]

where \( A \) is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful. The input parameter \( \text{uplo} \) indicates which part of the matrix is used. The function would leave other part untouched.

The user has to call \text{potrf} first to factorize matrix \( A \). If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_LOWER}, \( A \) is lower triangular Cholesky factor \( L \) corresponding to \( A = L^*L^H \).

If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_UPPER}, \( A \) is upper triangular Cholesky factor \( U \) corresponding to \( A = U^H*U \).

The operation is in-place, i.e. matrix \( X \) overwrites matrix \( B \) with the same leading dimension \( \text{ldb} \).

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

**API of potrs**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolveDN library context.</td>
</tr>
<tr>
<td>( \text{uplo} )</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix ( A ) lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>( n )</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{nrhs} )</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix ( X ) and ( B ).</td>
</tr>
<tr>
<td>( A )</td>
<td>device</td>
<td>input</td>
<td>&lt;type&gt; array of dimension ( \text{lda} * n ) with ( \text{lda} ) is not less than ( \max(1,n) ). ( A ) is either lower cholesky factor ( L ) or upper Cholesky factor ( U ).</td>
</tr>
<tr>
<td>( \text{lda} )</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>( B )</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension ( \text{ldb} * \text{nrhs} ). ( \text{ldb} ) is not less than ( \max(1,n) ). ( B ) is right hand side matrix. As an output, ( B ) is the solution matrix.</td>
</tr>
<tr>
<td>( \text{devInfo} )</td>
<td>device</td>
<td>output</td>
<td>if ( \text{devInfo} = 0 ), the Cholesky factorization is successful. if ( \text{devInfo} = -i ), the ( i )-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**
  - the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**
  - the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**
  - invalid parameters were passed \( [n<0, \text{nrhs}<0, \text{lda}<\max(1,n) \text{ or } \text{ldb}<\max(1,n)] \).
CUSOLVER_STATUS_ARCH_MISMATCH | the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR | an internal operation failed.

2.4.2.4. cusolverDnPotrs()[DEPRECATED]

[[DEPRECATED]] use cusolverDnXpotrs() instead. The routine will be removed in the next major release.

cusolverStatus_t

cusolverDnPotrs(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasFillMode_t uplo,
    int64_t n,
    int64_t nrhs,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeB,
    void *B,
    int64_t ldb,
    int *info)

This function solves a system of linear equations

\[ A^* X = B \]

where \( A \) is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful using the generic API interface. The input parameter \( \text{uplo} \) indicates which part of the matrix is used. The function would leave other part untouched.

The user has to call cusolverDnPotrf first to factorize matrix \( A \). If input parameter \( \text{uplo} \) is 
CUBLAS_FILL_MODE_LOWER, \( A \) is lower triangular Cholesky factor \( L \) corresponding to \( A = L^* L^H \).

If input parameter \( \text{uplo} \) is CUBLAS_FILL_MODE_UPPER, \( A \) is upper triangular Cholesky factor \( U \) corresponding to \( A = U^H U \).

The operation is in-place, i.e. matrix \( X \) overwrites matrix \( B \) with the same leading dimension \( \text{ldb} \).

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

Currently, cusolverDnPotrs supports only the default algorithm.

Table of algorithms supported by cusolverDnPotrs

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for cusolverDnPotrs:

API of potrs

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolveDN library context.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>params</th>
<th>host</th>
<th>input</th>
<th>structure with information collected by cusolverDnSetAdvOptions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix $A$ lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $X$ and $B$.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>input</td>
<td>array of dimension $lda \times n$ with $lda$ is not less than $\max(1,n)$. $A$ is either lower cholesky factor $L$ or upper Cholesky factor $U$.</td>
</tr>
<tr>
<td>$lda$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>dataTypeB</td>
<td>host</td>
<td>in</td>
<td>data type of array $B$.</td>
</tr>
<tr>
<td>$B$</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension $ldb \times nrhs$. $ldb$ is not less than $\max(1,n)$. As an input, $B$ is right hand side matrix. As an output, $B$ is the solution matrix.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if $info = 0$, the Cholesky factorization is successful. if $info = -i$, the $i$-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, `dataTypeA` is data type of the matrix $A$, `dataTypeB` is data type of the matrix $B$. `cusolverDnPotrs` only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>dataTypeA</th>
<th>dataTypeB</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRS</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRS</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRS</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRS</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS** the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED** the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE** invalid parameters were passed ($n<0$, $nrhs<0$, $lda<\max(1,n)$ or $ldb<\max(1,n)$).
- **CUSOLVER_STATUS_ARCH_MISMATCH** the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR** an internal operation failed.
2.4.2.5. \texttt{cusolverDn<t>potri()}

These helper functions calculate the necessary size of work buffers.

\begin{verbatim}
cusolverStatus_t
cusolverDnSpotri_bufferSize(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    int *Lwork );

cusolverStatus_t
cusolverDnDpotri_bufferSize(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    int *Lwork );

cusolverStatus_t
cusolverDnCpotri_bufferSize(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    int *Lwork );

cusolverStatus_t
cusolverDnZpotri_bufferSize(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    int *Lwork );
\end{verbatim}

The S and D data types are real valued single and double precision, respectively.

\begin{verbatim}
cusolverStatus_t
cusolverDnSpotri(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    float *Workspace,
    int Lwork,
    int *devInfo );

cusolverStatus_t
cusolverDnDpotri(cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *Workspace,
    int Lwork,
    int *devInfo );
\end{verbatim}
The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
 cusolerDnCpotri(cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    cuComplex *Workspace,
    int Lwork,
    int *devInfo);

cusolverStatus_t
 cusolerDnZpotri(cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    cuDoubleComplex *Workspace,
    int Lwork,
    int *devInfo);

This function computes the inverse of a positive-definite matrix $A$ using the Cholesky factorization

$$A = L^* L^H = U^* U$$

computed by $\text{potrf}()$.

$A$ is an $n \times n$ matrix containing the triangular factor $L$ or $U$ computed by the Cholesky factorization. Only lower or upper part is meaningful and the input parameter $\text{uplo}$ indicates which part of the matrix is used. The function would leave the other part untouched.

If the input parameter $\text{uplo}$ is CUBLAS_FILL_MODE_LOWER, only lower triangular part of $A$ is processed, and replaced the by lower triangular part of the inverse of $A$.

If the input parameter $\text{uplo}$ is CUBLAS_FILL_MODE_UPPER, only upper triangular part of $A$ is processed, and replaced by the upper triangular part of the inverse of $A$.

The user has to provide the working space which is pointed to by input parameter $\text{Workspace}$. The input parameter $\text{Lwork}$ is the size of the working space, returned by $\text{potri_bufferSize}()$.

If the computation of the inverse fails, i.e. some leading minor of $L$ or $U$, is null, the output parameter $\text{devInfo}$ would indicate the smallest leading minor of $L$ or $U$ which is not positive definite.

If the output parameter $\text{devInfo} = -i$ (less than zero), the $i$-th parameter is wrong (not counting the handle).

**API of $\text{potri}$**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix $A$ lower or upper part is stored, the other part is not referenced.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>n</th>
<th>host</th>
<th>input</th>
<th>number of rows and columns of matrix A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>where lda is not less than max(1,n).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>Workspace</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size Lwork.</td>
</tr>
<tr>
<td>Lwork</td>
<td>host</td>
<td>input</td>
<td>size of Workspace, returned by potri_bufferSize.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the computation of the inverse is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo = i, the leading minor of order i is zero.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.2.6. cusolverDn&lt;t&gt;getrf()

These helper functions calculate the size of work buffers needed.

```c
cusolverStatus_t
cusolverDnSgetrf_bufferSize(cusolverDnHandle_t handle,
    int m,
    int n,
    float *A,
    int lda,
    int *Lwork );

cusolverStatus_t
cusolverDnDgetrf_bufferSize(cusolverDnHandle_t handle,
    int m,
    int n,
    double *A,
    int lda,
    int *Lwork );

cusolverStatus_t
cusolverDnCgetrf_bufferSize(cusolverDnHandle_t handle,
    int m,
    int n,
    cuComplex *A,
    int lda,
    int *Lwork );
```
Using the CUSOLVER API

The S and D data types are real single and double precision, respectively.

```
cusolverStatus_t
cusolverDnSgetrf(cusolverDnHandle_t handle, int m, int n, float *A, int lda, float *Workspace, int *devIpiv, int *devInfo);
```

```
cusolverStatus_t
cusolverDnDgetrf(cusolverDnHandle_t handle, int m, int n, double *A, int lda, double *Workspace, int *devIpiv, int *devInfo);
```

The C and Z data types are complex valued single and double precision, respectively.

```
cusolverStatus_t
cusolverDnCgetrf(cusolverDnHandle_t handle, int m, int n, cuComplex *A, int lda, cuComplex *Workspace, int *devIpiv, int *devInfo);
```

```
cusolverStatus_t
cusolverDnZgetrf(cusolverDnHandle_t handle, int m, int n, cuDoubleComplex *A, int lda, cuDoubleComplex *Workspace, int *devIpiv, int *devInfo);
```

This function computes the LU factorization of a \( m \times n \) matrix

\[
P \cdot A = L \cdot U
\]

where \( A \) is a \( m \times n \) matrix, \( P \) is a permutation matrix, \( L \) is a lower triangular matrix with unit diagonal, and \( U \) is an upper triangular matrix.
The user has to provide working space which is pointed by input parameter Workspace. The input parameter Lwork is size of the working space, and it is returned by getrf_bufferSize().

If LU factorization failed, i.e. matrix $A \ [U]$ is singular, The output parameter devInfo=i indicates $U(i,i) = 0$.

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle).

If devIpiv is null, no pivoting is performed. The factorization is $A=L*U$, which is not numerically stable.

No matter LU factorization failed or not, the output parameter devIpiv contains pivoting sequence, row $i$ is interchanged with row devIpiv(i).

The user can combine getrf and getrs to complete a linear solver. Please refer to appendix D.1.

Remark: getrf uses fastest implementation with large workspace of size $m*n$. The user can choose the legacy implementation with minimal workspace by Getrf and cusolverDnSetAdvOptions(params, CUSOLVERDN_GETRF, CUSOLVER_ALG_1).

**API of getrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension $lda * n$ with $lda$ is not less than $\max(1,m)$.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>Workspace</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size Lwork.</td>
</tr>
<tr>
<td>devIpiv</td>
<td>device</td>
<td>output</td>
<td>array of size at least $\min(m,n)$, containing pivot indices.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the LU factorization is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo = i, the $U(i,i) = 0$.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [$m,n&lt;0$ or $lda&lt;\max(1,m)$].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
2.4.2.7. cusolverDnGetrf() [DEPRECATED]

[[DEPRECATED]] use cusolverDnXgetrf() instead. The routine will be removed in the next major release.

The helper function below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
cusolverDnGetrf_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType computeType,
    size_t *workspaceInBytes )
```

The function below

```c
 cusolverStatus_t
cusolverDnGetrf(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    int64_t *ipiv,
    cudaDataType computeType,
    void *pBuffer,
    size_t workspaceInBytes,
    int *info )
```

computes the LU factorization of a \( m \times n \) matrix

\[
P^* A = L^* U
\]

where \( A \) is a \( m \times n \) matrix, \( P \) is a permutation matrix, \( L \) is a lower triangular matrix with unit diagonal, and \( U \) is an upper triangular matrix using the generic API interface.

If LU factorization failed, i.e. matrix \( A \) \( [U] \) is singular, The output parameter \( info = i \) indicates \( U(i,i) = 0 \).

If output parameter \( info = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

If \( ipiv \) is null, no pivoting is performed. The factorization is \( A = L^* U \), which is not numerically stable.

No matter LU factorization failed or not, the output parameter \( ipiv \) contains pivoting sequence, row \( i \) is interchanged with row \( ipiv(i) \).
The user has to provide working space which is pointed by input parameter pBuffer. The input parameter workspaceInBytes is size in bytes of the working space, and it is returned by cusolverDnGetrf_bufferSize().

The user can combine cusolverDnGetrf and cusolverDnGetrs to complete a linear solver.

Currently, cusolverDnGetrf supports two algorithms. To select legacy implementation, the user has to call cusolverDnSetAdvOptions.

### Table of algorithms supported by cusolverDnGetrf

<table>
<thead>
<tr>
<th>CUSOLVER_ALG_0 or NULL</th>
<th>Default algorithm. The fastest, requires a large workspace of m*n elements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_ALG_1</td>
<td>Legacy implementation</td>
</tr>
</tbody>
</table>

### List of input arguments for cusolverDnGetrf_bufferSize and cusolverDnGetrf:

**API of cusolverDnGetrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n with lda is not less than max(1,m).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>ipiv</td>
<td>device</td>
<td>output</td>
<td>array of size at least min(m,n), containing pivot indices.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>in/out</td>
<td>Working space. Array of type void of size workspaceInBytes bytes.</td>
</tr>
<tr>
<td>workspaceInBytes</td>
<td>host</td>
<td>input</td>
<td>size in bytes of pBuffer, returned by cusolverDnGetrf_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the LU factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i, the U(i,i) = 0.</td>
</tr>
</tbody>
</table>

The generic API has two different types, dataTypeA is data type of the matrix A, computeType is compute type of the operation. cusolverDnGetrf only supports the following four combinations.

### valid combination of data type and compute type

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGETRF</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGETRF</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

### cuSOLVER Library

<table>
<thead>
<tr>
<th>CUDA_C_32F</th>
<th>CUDA_C_64F</th>
<th>CGETRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGETRF</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (m, n&lt;0 or lda&lt;max(1,m)).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.2.8. cusolverDn<t>getrs()

```c
cusolverStatus_t
cusolverDnSgetrs(cusolverDnHandle_t handle,  
cublasOperation_t trans,  
int n,  
int nrhs,  
const float *A,  
int lda,  
const int *devIpiv,  
float *B,  
int ldb,  
int *devInfo );
```

```c
cusolverStatus_t
cusolverDnDgetrs(cusolverDnHandle_t handle,  
cublasOperation_t trans,  
int n,  
int nrhs,  
const double *A,  
int lda,  
const int *devIpiv,  
double *B,  
int ldb,  
int *devInfo );
```

```c
cusolverStatus_t
cusolverDnCgetrs(cusolverDnHandle_t handle,  
cublasOperation_t trans,  
int n,  
int nrhs,  
const cuComplex *A,  
int lda,  
const int *devIpiv,  
cuComplex *B,  
int ldb,  
int *devInfo );
```

```c
cusolverStatus_t
cusolverDnZgetrs(cusolverDnHandle_t handle,  
cublasOperation_t trans,  
int n,  
int nrhs,  
const cuComplex *A,  
int lda,  
const int *devIpiv,  
cuComplex *B,  
int ldb,  
int *devInfo );
```
This function solves a linear system of multiple right-hand sides

\[ \text{op}(A)X = B \]

where \( A \) is an \( n \times n \) matrix, and was LU-factored by \( \text{getrf} \), that is, lower triangular part of \( A \) is \( L \), and upper triangular part (including diagonal elements) of \( A \) is \( U \). \( B \) is a \( n \times nrhs \) right-hand side matrix.

The input parameter \( \text{trans} \) is defined by

\[
\text{op}(A) = \begin{cases} 
A & \text{if } \text{trans} == \text{CUBLAS}_\text{OP}_\text{N} \\
A^T & \text{if } \text{trans} == \text{CUBLAS}_\text{OP}_\text{T} \\
A^H & \text{if } \text{trans} == \text{CUBLAS}_\text{OP}_\text{C} 
\end{cases}
\]

The input parameter \( \text{devIpiv} \) is an output of \( \text{getrf} \). It contains pivot indices, which are used to permute right-hand sides.

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

The user can combine \( \text{getrf} \) and \( \text{getrs} \) to complete a linear solver. Please refer to appendix D.1.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>trans</td>
<td>host</td>
<td>input</td>
<td>operation ( \text{op}(A) ) that is non- or [conj.] transpose.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of right-hand sides.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>input</td>
<td>&lt;type&gt; array of dimension ( lda \times n ) with ( lda ) is not less than ( \max(1,n) ).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>devIpiv</td>
<td>device</td>
<td>input</td>
<td>array of size at least ( n ), containing pivot indices.</td>
</tr>
<tr>
<td>B</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ( ldb \times nrhs ) with ( ldb ) is not less than ( \max(1,n) ).</td>
</tr>
<tr>
<td>ldb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( B ).</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if ( \text{devInfo} = 0 ), the operation is successful. if ( \text{devInfo} = -i ), the ( i )-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>
Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( n &lt; 0 ) or ( \text{lda} &lt; \max(1,n) ) or ( \text{ldb} &lt; \max(1,n) ).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.2.9. cusolverDnGetrs() [DEPRECATED]

[DEPRECATED] use cusolverDnXgetrs() instead. The routine will be removed in the next major release.

cusolverStatus_t

```c
    cusolverDnGetrs(
        cusolverDnHandle_t handle,
        cusolverDnParams_t params,
        cublasOperation_t trans,
        int64_t n,
        int64_t nrhs,
        cudaDataType dataTypeA,
        const void *A,
        int64_t lda,
        const int64_t *ipiv,
        cudaDataType dataTypeB,
        void *B,
        int64_t ldb,
        int *info
    )
```

This function solves a linear system of multiple right-hand sides

\[ \text{op}(A)X = B \]

where \( A \) is a \( n \times n \) matrix, and was LU-factored by cusolverDnGetrf, that is, lower triangular part of \( A \) is \( L \), and upper triangular part (including diagonal elements) of \( A \) is \( U \). \( B \) is a \( n \times nrhs \) right-hand side matrix using the generic API interface.

The input parameter `trans` is defined by

\[
\text{op}(A) = \begin{cases} 
    A & \text{if } \text{trans} == \text{CUBLAS_OP_N} \\
    A^T & \text{if } \text{trans} == \text{CUBLAS_OP_T} \\
    A^H & \text{if } \text{trans} == \text{CUBLAS_OP_C} 
\end{cases}
\]

The input parameter `ipiv` is an output of cusolverDnGetrf. It contains pivot indices, which are used to permute right-hand sides.

If output parameter `info` = -1 (less than zero), the \( i \)-th parameter is wrong (not counting handle).

The user can combine cusolverDnGetrf and cusolverDnGetrs to complete a linear solver.

Currently, cusolverDnGetrs supports only the default algorithm.

Table of algorithms supported by cusolverDnGetrs

CuSOLVER Library
Using the CUSOLVER API

CUSOLVER_ALG_0 or NULL
Default algorithm.

List of input arguments for cusolverDnGetrs:

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>trans</td>
<td>host</td>
<td>input</td>
<td>operation ( \text{op}(A) ) that is non- or (conj.) transpose.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of right-hand sides.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array ( A ).</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>input</td>
<td>array of dimension ( \text{lda} \times n ) with ( \text{lda} ) not less than ( \max(1,n) ).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>ipiv</td>
<td>device</td>
<td>input</td>
<td>array of size at least ( n ), containing pivot indices.</td>
</tr>
<tr>
<td>dataTypeB</td>
<td>host</td>
<td>in</td>
<td>data type of array ( B ).</td>
</tr>
<tr>
<td>B</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ( \text{ldb} \times \text{nrhs} ) with ( \text{ldb} ) not less than ( \max(1,n) ).</td>
</tr>
<tr>
<td>ldb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( B ).</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if ( \text{info} = 0 ), the operation is successful. If ( \text{info} = -i ), the ( i )-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, \( \text{dataTypeA} \) is data type of the matrix \( A \) and \( \text{dataTypeB} \) is data type of the matrix \( B \). **cusolverDnGetrs** only supports the following four combinations.

### Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>( \text{DataTypeA} )</th>
<th>( \text{DataTypeB} )</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGETRS</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGETRS</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGETRS</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGETRS</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ([n&lt;0 \text{ or } \text{lda}&lt;\max(1,n) \text{ or } \text{ldb}&lt;\max(1,n)]).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_INTERNAL_ERROR | an internal operation failed.

2.4.2.10. cusolverDn<t1><t2>gesv()

These functions are modelled after functions DSGESV and ZCGESV from LAPACK. They compute the solution of a system of linear equations with one or multiple right hand sides using mixed precision iterative refinement techniques based on the LU factorization Xgesv. These functions are similar in term of functionalities to the full precision LU solver (Xgesv, where X denotes Z,C,D,S) but it uses lower precision internally in order to provide faster time to solution, from here comes the name mixed precision. Mixed precision iterative refinement techniques means that the solver compute an LU factorization in lower precision and then iteratively refine the solution to achieve the accuracy of the Inputs/Outputs datatype precision. The <t1> corresponds to the Inputs/Outputs datatype precision while <t2> represent the internal lower precision at which the factorization will be carried on.

\[ A \times X = B \]

Where \( A \) is \( n \)-by-\( n \) matrix and \( X \) and \( B \) are \( n \)-by-\( nrhs \) matrices.

Functions API are designed to be as close as possible to LAPACK API to be considered as a quick and easy drop-in replacement. Parameters and behavior are mostly the same as LAPACK counterparts. Description of these functions and differences from LAPACK is given below. <t1><t2>gesv() functions are designated by two floating point precisions The <t1> corresponds to the main precision (e.g., Inputs/Outputs datatype precision) and the <t2> represent the internal lower precision at which the factorization will be carried on. cusolver<t1><t2>gesv() first attempts to factorize the matrix in lower precision and use this factorization within an iterative refinement procedure to obtain a solution with same normwise backward error as the main precision <t1>. If the approach fails to converge, then the method fallback to the main precision factorization and solve (Xgesv) such a way that there is always a good solution at the output of these functions. If <t2> is equal to <t1>, then it is not a mixed precision process but rather a full one precision factorisation, solve and refinement within the same main precision.

The iterative refinement process is stopped if

\[ ITER > ITERMAX \]

or for all the RHS we have:

\[ RNRM < \sqrt{N} \times XNRM \times ANRM \times EPS \times BWDMAX \]

where

- ITER is the number of the current iteration in the iterative refinement process
- RNRM is the infinity-norm of the residual
- XNRM is the infinity-norm of the solution
- ANRM is the infinity-operator-norm of the matrix A
- EPS is the machine epsilon that matches LAPACK <t1>LAMCH('Epsilon')
The value ITERMAX and BWDMAX are fixed to 50 and 1.0 respectively.

The function returns value describes the results of the solving process. A CUSOLVER_STATUS_SUCCESS indicates that the function finished with success otherwise, it indicates if one of the API arguments is incorrect, or if the function did not finish with success. More details about the error will be in the nIter and the dinfo API parameters. See their description below for more details. User should provide the required workspace allocated on device memory. The amount of bytes required can be queried by calling the respective function gesv_bufferSize() for each of the available function.

Note that in addition to the two mixed precision functions available in LAPACK (e.g., dsgesv and zcgesv), we provide a large set of mixed precision functions that include half, bfloat and tensorfloat as a lower precision as well as same precision functions (e.g., main and lowest precision are equal if <t2> is equal to <t1>). The following table specifies which precisions will be used for which interface function.

Tensor Float (TF32), introduced with NVIDIA Ampere Architecture GPUs, is the most robust tensor core accelerated compute mode for the iterative refinement solver. It is able to solve the widest range of problems in HPC arising from different applications and provides up to 4X and 5X speedup for real and complex systems, respectively. On Volta and Turing architecture GPUs, half precision tensor core acceleration is recommended. In cases where the iterative refinement solver fails to converge to the desired accuracy (main precision, INOUT data precision), it is recommended to use main precision as internal lowest precision (i.e., cusolverDn[DD, ZZ]gesv for the FP64 case).

### Supported combinations of floating point precisions for cusolver <t1><t2>gesv() functions

<table>
<thead>
<tr>
<th>Interface function</th>
<th>Main precision (matrix, rhs and solution datatype)</th>
<th>Lowest precision allowed to be used internally</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusolverDnZZgesv</td>
<td>cuDoubleComplex</td>
<td>double complex</td>
</tr>
<tr>
<td>cusolverDnZCgesv *has LAPACK counterparts</td>
<td>cuDoubleComplex</td>
<td>single complex</td>
</tr>
<tr>
<td>cusolverDnZKgesv</td>
<td>cuDoubleComplex</td>
<td>half complex</td>
</tr>
<tr>
<td>cusolverDnZYgesv</td>
<td>cuDoubleComplex</td>
<td>bfloat complex</td>
</tr>
<tr>
<td>cusolverDnCCgesv</td>
<td>cuComplex</td>
<td>single complex</td>
</tr>
<tr>
<td>cusolverDnCKgesv</td>
<td>cuComplex</td>
<td>half complex</td>
</tr>
<tr>
<td>cusolverDnCEgesv</td>
<td>cuComplex</td>
<td>bfloat complex</td>
</tr>
<tr>
<td>cusolverDnCYgesv</td>
<td>cuComplex</td>
<td>tensorfloat complex</td>
</tr>
<tr>
<td>cusolverDnDDgesv</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>cusolverDnDSgesv *has LAPACK counterparts</td>
<td>double</td>
<td>single</td>
</tr>
<tr>
<td>cusolverDnDHgesv</td>
<td>double</td>
<td>half</td>
</tr>
<tr>
<td>cusolverDnDBgesv</td>
<td>double</td>
<td>bfloat</td>
</tr>
<tr>
<td>cusolverDnDXgesv</td>
<td>double</td>
<td>tensorfloat</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusolverDnSSgesv</td>
<td>float</td>
<td>single</td>
</tr>
<tr>
<td>cusolverDnSHgesv</td>
<td>float</td>
<td>half</td>
</tr>
<tr>
<td>cusolverDnSBgesv</td>
<td>float</td>
<td>bfloat</td>
</tr>
<tr>
<td>cusolverDnSXgesv</td>
<td>float</td>
<td>tensorfloat</td>
</tr>
</tbody>
</table>

cusolverDn<t1><t2>gesv_bufferSize() functions will return workspace buffer size in bytes required for the corresponding cusolverDn<t1><t2>gesv() function.

cusolverStatus_t

cusolverDnZZgesv_bufferSize(  
    cusolverHandle_t          handle,  
    int                      n,  
    int                      nrhs,  
    cuDoubleComplex          * dA,  
    int                      ldda,  
    int                      * dipiv,  
    cuDoubleComplex          * dB,  
    int                      lddb,  
    cuDoubleComplex          * dX,  
    int                      lddx,  
    void                     * dwork,  
    size_t                    * lwork_bytes);

cusolverStatus_t

cusolverDnZCgesv_bufferSize(  
    cusolverHandle_t          handle,  
    int                      n,  
    int                      nrhs,  
    cuDoubleComplex          * dA,  
    int                      ldda,  
    int                      * dipiv,  
    cuDoubleComplex          * dB,  
    int                      lddb,  
    cuDoubleComplex          * dX,  
    int                      lddx,  
    void                     * dwork,  
    size_t                    * lwork_bytes);

cusolverStatus_t

cusolverDnZKgesv_bufferSize(  
    cusolverHandle_t          handle,  
    int                      n,  
    int                      nrhs,  
    cuDoubleComplex          * dA,  
    int                      ldda,  
    int                      * dipiv,  
    cuDoubleComplex          * dB,  
    int                      lddb,  
    cuDoubleComplex          * dX,  
    int                      lddx,  
    void                     * dwork,  
    size_t                    * lwork_bytes);

cusolverStatus_t

cusolverDnZEgesv_bufferSize(  
    cusolverHandle_t          handle,  
    int                      n,  
    int                      nrhs,  
    int                      * lwork_bytes);
Using the CUSOLVER API

```c
#include <cusolverDn.h>

// cuDoubleComplex * dA,
int lda,
int * dipiv,
int * dB,
int * dX,
int ldda,
int lddb,
int lddx,
void * dwork,
size_t * lwork_bytes);

cusolverStatus_t
cusolverDnZYgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int ldda,
    int * dipiv,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);

cusolverStatus_t
cusolverDnCCgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    cuComplex * dA,
    int ldda,
    int * dipiv,
    cuComplex * dB,
    int lddb,
    cuComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);

cusolverStatus_t
cusolverDnCKgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    cuComplex * dA,
    int ldda,
    int * dipiv,
    cuComplex * dB,
    int lddb,
    cuComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);

cusolverStatus_t
cusolverDnCEgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    cuComplex * dA,
    int ldda,
    int * dipiv,
    cuComplex * dB,
    int lddb,
    cuComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);
```

---

cuSOLVER Library
Using the CUSOLVER API

```
cusolverStatus_t

cusolverDnCYgesv_bufferSize(
cusolverHandle_t handle,
int n,
int nrhs,
cuComplex * dA,
int ldda,
int * dipiv,
double * dB,
int lddb,
cuComplex * dx,
int lddx,
void * dwork,
size_t * lwork_bytes);

cusolverStatus_t

cusolverDnDDgesv_bufferSize(
cusolverHandle_t handle,
int n,
int nrhs,
double * dA,
int ldda,
int * dipiv,
double * dB,
int lddb,
double * dx,
int lddx,
void * dwork,
size_t * lwork_bytes);

cusolverStatus_t

cusolverDnDSgesv_bufferSize(
cusolverHandle_t handle,
int n,
int nrhs,
double * dA,
int ldda,
int * dipiv,
double * dB,
int lddb,
double * dx,
int lddx,
void * dwork,
size_t * lwork_bytes);

cusolverStatus_t

cusolverDnDHgesv_bufferSize(
cusolverHandle_t handle,
int n,
int nrhs,
double * dA,
int ldda,
int * dipiv,
double * dB,
int lddb,
double * dx,
int lddx,
void * dwork,
size_t * lwork_bytes);
```
Using the CUSOLVER API

```c
int *   dipiv,
double *   dB,
int lddb,
double *   dx,
int lddx,
void *   dwork,
size_t lwork_bytes);
```

cusolverStatus_t
cusolverDnDBgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    double *   dA,
    int ldda,
    int *   dipiv,
    double *   dB,
    int lddb,
    double *   dx,
    int lddx,
    void *   dwork,
    size_t lwork_bytes);

cusolverStatus_t
cusolverDnDXgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    double *   dA,
    int ldda,
    int *   dipiv,
    double *   dB,
    int lddb,
    double *   dx,
    int lddx,
    void *   dwork,
    size_t lwork_bytes);

cusolverStatus_t
cusolverDnSSgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    float *   dA,
    int ldda,
    int *   dipiv,
    float *   dB,
    int lddb,
    float *   dx,
    int lddx,
    void *   dwork,
    size_t lwork_bytes);

cusolverStatus_t
cusolverDnSHgesv_bufferSize(
    cusolverHandle_t handle,
    int n,
    int nrhs,
    float *   dA,
    int ldda,
    int *   dipiv,
```
**Parameters of cusolverDn<T1><T2>gesv_bufferSize() functions**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDN library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of rows and columns of square matrix A. Should be non-negative.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>None</td>
<td>Matrix A with size n-by-n. Can be NULL.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. lda &gt;= n.</td>
</tr>
<tr>
<td>dipiv</td>
<td>device</td>
<td>None</td>
<td>Pivoting sequence. Not used and can be NULL.</td>
</tr>
<tr>
<td>dB</td>
<td>device</td>
<td>None</td>
<td>Set of right hand sides B of size n-by-nrhs. Can be NULL.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>lddb</th>
<th>host</th>
<th>input</th>
<th>leading dimension of two-dimensional array used to store matrix of right hand sides B. ( ldb \geq n ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>dX</td>
<td>device</td>
<td>None</td>
<td>Set of solution vectors ( X ) of size ( n )-by-( nrhs ). Can be NULL.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of solution vectors ( X ). ( ldx \geq n ).</td>
</tr>
<tr>
<td>dwork</td>
<td>device</td>
<td>none</td>
<td>Pointer to device workspace. Not used and can be NULL.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>output</td>
<td>Pointer to a variable where required size of temporary workspace in bytes will be stored. Can’t be NULL.</td>
</tr>
</tbody>
</table>

```c

cusolverStatus_t cusolverDnZZgesv(  
cusolverDnHandle_t      handle,  
int                     n,  
int                     nrhs,  
cuDoubleComplex     *   dA,  
int                     ldda,  
int                 *   dipiv,  
cuDoubleComplex     *   dB,  
int                     lddb,  
cuDoubleComplex     *   dX,  
int                     lddx,  
void                *   dWorkspace,  
size_t                  lwork_bytes,  
int                 *   niter,  
int                 *   dinfo);

cusolverStatus_t cusolverDnZCgesv(  
cusolverDnHandle_t      handle,  
int                     n,  
int                     nrhs,  
cuDoubleComplex     *   dA,  
int                     ldda,  
int                 *   dipiv,  
cuDoubleComplex     *   dB,  
int                     lddb,  
cuDoubleComplex     *   dX,  
int                     lddx,  
void                *   dWorkspace,  
size_t                  lwork_bytes,  
int                 *   niter,  
int                 *   dinfo);

cusolverStatus_t cusolverDnZKgesv(  
cusolverDnHandle_t      handle,  
int                     n,  
int                     nrhs,  
cuDoubleComplex     *   dA,  
int                     ldda,  
int                 *   dipiv,  
cuDoubleComplex     *   dB,  
int                     lddb,  
cuDoubleComplex     *   dX,  
int                     lddx,  
void                *   dWorkspace,  
size_t                  lwork_bytes,  
int                 *   niter,  
int                 *   dinfo);
```
Using the CUSOLVER API

```
int                     lddx,
void                *   dWorkspace,
size_t                  lwork_bytes,
int                 *   niter,
int                 *   dinfo);

cusolverStatus_t cusolverDnZEgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuDoubleComplex     *   dA,
    int                     lda,
    int                 *   dipiv,
    cuDoubleComplex     *   dB,
    int                     ldb,
    cuDoubleComplex     *   dX,
    int                     ldx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnZYgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuDoubleComplex     *   dA,
    int                     lda,
    int                 *   dipiv,
    cuDoubleComplex     *   dB,
    int                     ldb,
    cuDoubleComplex     *   dX,
    int                     ldx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnCCgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuComplex           *   dA,
    int                     lda,
    int                 *   dipiv,
    cuComplex           *   dB,
    int                     ldb,
    cuComplex           *   dX,
    int                     ldx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnCKgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuComplex           *   dA,
    int                     lda,
    int                 *   dipiv,
```
Using the CUSOLVER API

cusolverStatus_t cusolverDnCEgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuComplex           *   dA,
    int                     ldda,
    int                     *   dipiv,
    cuComplex           *   dB,
    int                     lddb,
    cuComplex           *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnCYgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    cuComplex           *   dA,
    int                     ldda,
    int                     *   dipiv,
    cuComplex           *   dB,
    int                     lddb,
    cuComplex           *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnDDgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
    double              *   dA,
    int                     ldda,
    int                     *   dipiv,
    double              *   dB,
    int                     lddb,
    double              *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                lwork_bytes,
    int                 *   niter,
    int                 *   dinfo);

cusolverStatus_t cusolverDnDSgesv(
    cusolverDnHandle_t      handle,
    int                     n,
    int                     nrhs,
double * dA,
int ldda,
int * dipiv,
double * dB,
int ldddb,
double * dX,
int lddx,
void * dWorkspace,
size_t lwork_bytes,
int * niter,
int * dinfo);

cusolverStatus_t cusolverDnDHgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
double * dA,
    int ldda,
    int * dipiv,
double * dB,
    int ldddb,
double * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnDBgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
double * dA,
    int ldda,
    int * dipiv,
double * dB,
    int ldddb,
double * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnDXgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
double * dA,
    int ldda,
    int * dipiv,
double * dB,
    int ldddb,
double * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnSSgesv(
Using the CUSOLVER API

cusolverDnHandle_t handle,
int n,
int nrhs,
float * dA,
int ldda,
int * dipiv,
float * dB,
int lddb,
float * dX,
int lddx,
void * dWorkspace,
size_t lwork_bytes,
int * niter,
int * dinfo);

cusolverStatus_t cusolverDnSHgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
    float * dA,
    int ldda,
    int * dipiv,
    float * dB,
    int lddb,
    float * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnSBgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
    float * dA,
    int ldda,
    int * dipiv,
    float * dB,
    int lddb,
    float * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnSXgesv(
    cusolverDnHandle_t handle,
    int n,
    int nrhs,
    float * dA,
    int ldda,
    int * dipiv,
    float * dB,
    int lddb,
    float * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
Parameters of cusolverDn<T1><T2>gesv() functions

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDN library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of rows and columns of square matrix A. Should be non-negative.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>in/out</td>
<td>Matrix A with size n-by-n. Can’t be NULL. On return - unchanged if the iterative refinement process converged. If not - will contains the factorization of the matrix A in the main precision &lt;T1&gt; (A = P * L * U, where P - permutation matrix defined by vector ipiv, L and U - lower and upper triangular matrices).</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. lda &gt;= n.</td>
</tr>
<tr>
<td>dipiv</td>
<td>device</td>
<td>output</td>
<td>Vector that defines permutation for the factorization - row i was interchanged with row ipiv[i].</td>
</tr>
<tr>
<td>dB</td>
<td>device</td>
<td>input</td>
<td>Set of right hand sides B of size n-by-nrhs. Can’t be NULL.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of right hand sides B. ldb &gt;= n.</td>
</tr>
<tr>
<td>dX</td>
<td>device</td>
<td>output</td>
<td>Set of solution vectors X of size n-by-nrhs. Can’t be NULL.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of solution vectors X. ldx &gt;= n.</td>
</tr>
<tr>
<td>dWorkspace</td>
<td>device</td>
<td>input</td>
<td>Pointer to an allocated workspace in device memory of size lwork_bytes.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>input</td>
<td>Size of the allocated device workspace. Should be at least what was returned by cusolverDn&lt;T1&gt;&lt;T2&gt;gesv_bufferSize() function</td>
</tr>
<tr>
<td>niter</td>
<td>host</td>
<td>output</td>
<td>If iter is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▶ &lt;0 : iterative refinement has failed, main precision [Inputs/Outputs precision] factorization has been performed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▶ -1 : taking into account machine parameters, n, nrhs, it is a priori not worth working in lower precision</td>
</tr>
</tbody>
</table>

```c
int * dinfo);
```
Using the CUSOLVER API

- -2 : overflow of an entry when moving from main to lower precision
- -3 : failure during the factorization
- -5 : overflow occurred during computation
- -50: solver stopped the iterative refinement after reaching maximum allowed iterations.
- >0 : iter is a number of iterations solver performed to reach convergence criteria.

<table>
<thead>
<tr>
<th>dinfo</th>
<th>device</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status of the IRS solver on the return. If 0 - solve was successful. If dinfo = -i then i-th argument is not valid. If dinfo = i, then $U(i,i)$ computed in main precision is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution could not be computed.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed, for example:</td>
</tr>
<tr>
<td></td>
<td>- n&lt;0</td>
</tr>
<tr>
<td></td>
<td>- lda&lt;max(1,n)</td>
</tr>
<tr>
<td></td>
<td>- ldb&lt;max(1,n)</td>
</tr>
<tr>
<td></td>
<td>- ldx&lt;max(1,n)</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the IRS solver supports compute capability 7.0 and above. The lowest precision options CUSOLVER_[CR]<em>16BF and CUSOLVER</em>[CR]_TF32 are only available on compute capability 8.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_WORKSPACE</td>
<td>lwork_bytes is smaller than the required workspace.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_OUT_OF_RANGE</td>
<td>Numerical error related to niters &lt;0, seeniters description for more details.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal error occurred, check the dinfo and the niters arguments for more details.</td>
</tr>
</tbody>
</table>

### 2.4.2.11. cusolverDnIRSXgesv()

This function is designed to perform same functionality as cusolverDn<T1><T2>gesv() functions, but wrapped in a more generic and expert interface that gives user more
control to parametrize the function as well as it provides more informations on output. cusolverDnIRSXgesv() allows additional control of the solver parameters such as setting:

- the main precision (Inputs/Outputs precision) of the solver,
- the lowest precision to be used internally by the solver,
- the refinement solver type,
- the maximum allowed number of iterations in the refinement phase,
- the tolerance of the refinement solver,
- the fallback to main precision,
- and more other.

through the configuration parameters structure gesv IRS params and its helper functions. For more details about what configuration can be set and its meaning please refer to all the functions in the cuSolverDN Helper Function Section that start with cusolverDnIRSParamsxxxx(). Moreover, cusolverDnIRSXgesv() provides additional informations on the output such as the convergence history (e.g., the residual norms) at each iteration and the number of iterations needed to converge. For more details about what informations can be retrieved and its meaning please refer to all the functions in the cuSolverDN Helper Function Section that start with cusolverDnIRSInfosxxxx().

The function returns value describes the results of the solving process. A CUSOLVER_STATUS_SUCCESS indicates that the function finished with success otherwise, it indicates if one of the API arguments is incorrect, or if the configurations of params/infos structure is incorrect or if the function did not finish with success. More details about the error can be found by checking the nIters and the dinfo API parameters. See their description below for further details. User should provide the required workspace allocated on device for the cusolverDnIRSXgesv() function. The amount of bytes required for the function can be queried by calling the respective function cusolverDnIRSXgesv_bufferSize(). Note that, if the user would like a particular configuration to be set via the params structure, it should be set before the call to cusolverDnIRSXgesv_bufferSize() to get the size of the required workspace.

Tensor Float (TF32), introduced with NVIDIA Ampere Architecture GPUs, is the most robust tensor core accelerated compute mode for the iterative refinement solver. It is able to solve the widest range of problems in HPC arising from different applications and provides up to 4X and 5X speedup for real and complex systems, respectively. On Volta and Turing architecture GPUs, half precision tensor core acceleration is recommended. In cases where the iterative refinement solver fails to converge to the desired accuracy (main precision, INOUT data precision), it is recommended to use main precision as internal lowest precision.

The following table provides all possible combinations values for the lowest precision corresponding to the Inputs/Outputs data type. Note that if the lowest precision matches the Inputs/Outputs datatype, then the main precision factorization will be used.

### Supported Inputs/Outputs data type and lower precision for the IRS solver

<table>
<thead>
<tr>
<th>Inputs/Outputs Data Type (e.g., main precision)</th>
<th>Supported values for the lowest precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_C_64F</td>
<td>CUSOLVER_C_64F, CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

CUSOLVER_C_32F
CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32

CUSOLVER_R_64F
CUSOLVER_R_64F, CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32

CUSOLVER_R_32F
CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32

The cusolverDnIRSXgesv_bufferSize() function returns the required workspace buffer size in bytes for the corresponding cusolverDnXgesv() call with the given gesv_irs_params configuration.

cusolverStatus_t cusolverDnIRSXgesv_bufferSize(
    cusolverDnHandle_t          handle,
    cusolverDnIRSParams_t       gesv_irs_params,
    cusolver_int_t              n,
    cusolver_int_t              nrhs,
    size_t                  *   lwork_bytes);

Parameters of cusolverDnIRSXgesv_bufferSize() functions

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDn library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>Xgesv configuration parameters</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of rows and columns of the square matrix A. Should be non-negative.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative. Note that, nrhs is limited to 1 if the selected IRS refinement solver is CUSOLVER_IRS_REFINE_GMRES, CUSOLVER_IRS_REFINE_GMRES_GMRES, CUSOLVER_IRS_REFINE_CLASSICAL_GMRES.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>out</td>
<td>Pointer to a variable, where the required size in bytes, of the workspace will be stored after a call to cusolverDnIRSXgesv_bufferSize. Can't be NULL.</td>
</tr>
</tbody>
</table>

cusolverStatus_t cusolverDnIRSXgesv(
    cusolverDnHandle_t          handle,
    cusolverDnIRSParams_t       gesv_irs_params,
    cusolverDnIRSInfos_t        gesv_irs_infos,
    int                         n,
    int                         nrhs,
    void                        *   dA,
    int                         lda,
    void                        *   dB,
    int                         ldb,
    void                        *   dX,
    int                         ldx,
    void                        *   dWorkspace,
```
size_t lwork_bytes,
int * dinfo);
```

**Parameters of cusolverDnIRSXgesv() functions**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDn library context.</td>
</tr>
<tr>
<td>gesv_irs_params</td>
<td>host</td>
<td>input</td>
<td>Configuration parameters structure, can serve one or more calls to any IRS solver</td>
</tr>
<tr>
<td>gesv_irs_infos</td>
<td>host</td>
<td>in/out</td>
<td>Info structure, where information about a particular solve will be stored. The gesv_irs_infos structure correspond to a particular call. Thus different calls requires different gesv_irs_infos structure otherwise, it will be overwritten.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of rows and columns of square matrix A. Should be non-negative.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative. Note that, nrhs is limited to 1 if the selected IRS refinement solver is CUSOLVER_IRS_REFINE_GMRES, CUSOLVER_IRS_REFINE_GMRES_GMRES, CUSOLVER_IRS_REFINE_CLASSICAL_GMRES.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>in/out</td>
<td>Matrix A with size n-by-n. Can’t be NULL. On return - will contain the factorization of the matrix A in the main precision (A = P * L * U, where P - permutation matrix defined by vector ipiv, L and U - lower and upper triangular matrices) if the iterative refinement solver was set to CUSOLVER_IRS_REFINE_NONE and the lowest precision is equal to the main precision (Inputs/Ouputs datatype), or if the iterative refinement solver did not converge and the fallback to main precision was enabled (fallback enabled is the default setting); - unchanged otherwise.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of two-dimensional array used to store matrix A. ldda &gt;= n.</td>
</tr>
<tr>
<td>dB</td>
<td>device</td>
<td>input</td>
<td>Set of right hand sides B of size n-by-nrhs. Can’t be NULL.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of two-dimensional array used to store matrix of right hand sides B. ldcb &gt;= n.</td>
</tr>
<tr>
<td>dX</td>
<td>device</td>
<td>output</td>
<td>Set of soultion vectors X of size n-by-nrhs. Can’t be NULL.</td>
</tr>
</tbody>
</table>

**DU-06709-001_v11.1 | 73**
### Using the CUSOLVER API

<table>
<thead>
<tr>
<th><strong>lddx</strong></th>
<th><strong>host</strong></th>
<th><strong>input</strong></th>
<th>leading dimension of two-dimensional array used to store matrix of solution vectors ( X ). ( ldx \geq n ).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dWorkspace</strong></td>
<td><strong>device</strong></td>
<td><strong>input</strong></td>
<td>Pointer to an allocated workspace in device memory of size ( lwork _bytes ).</td>
</tr>
<tr>
<td><strong>lwork _bytes</strong></td>
<td><strong>host</strong></td>
<td><strong>input</strong></td>
<td>Size of the allocated device workspace. Should be at least what was returned by <code>cusolverDnIRSxgesv_bufferSize()</code> function.</td>
</tr>
<tr>
<td><strong>niters</strong></td>
<td><strong>host</strong></td>
<td><strong>output</strong></td>
<td>If ( \text{iter} ) is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-&lt;0 : iterative refinement has failed, main precision (Inputs/Outputs precision) factorization has been performed if fallback is enabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1 : taking into account machine parameters, ( n, nrhs ), it is a priori not worth working in lower precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2 : overflow of an entry when moving from main to lower precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3 : failure during the factorization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-5 : overflow occurred during computation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-maxiter: solver stopped the iterative refinement after reaching maximum allowed iterations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&gt;0 : ( \text{iter} ) is a number of iterations solver performed to reach convergence criteria</td>
</tr>
<tr>
<td><strong>dinfo</strong></td>
<td><strong>device</strong></td>
<td><strong>output</strong></td>
<td>Status of the IRS solver on the return. If 0 - solve was successful. If ( dinfo ) = -( i ) then ( i )-th argument is not valid. If ( dinfo ) = ( i ), then ( U(i,i) ) computed in main precision is exactly zero. The factorization has been completed, but the factor ( U ) is exactly singular, so the solution could not be computed.</td>
</tr>
</tbody>
</table>

#### Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed, for example:</td>
</tr>
<tr>
<td></td>
<td>- ( n&lt;0 )</td>
</tr>
<tr>
<td></td>
<td>- ( lda&lt;\text{max}(1,n) )</td>
</tr>
<tr>
<td></td>
<td>- ( ldb&lt;\text{max}(1,n) )</td>
</tr>
<tr>
<td></td>
<td>- ( ldx&lt;\text{max}(1,n) )</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_ARCH_MISMATCH  
the IRS solver supports compute capability 7.0 and above. The lowest precision options CUSOLVER_[CR]_16BF and CUSOLVER_[CR]_TF32 are only available on compute capability 8.0 and above.

CUSOLVER_STATUS_INVALID_WORKSPACE  
lwork_bytes is smaller than the required workspace. Could happen if the users called cusolverDnIRSXgesv_bufferSize() function, then changed some of the configurations setting such as the lowest precision.

CUSOLVER_STATUS_IRS_OUT_OF_RANGE  
Numerical error related to niters <0, see niters description for more details.

CUSOLVER_STATUS_INTERNAL_ERROR  
an internal error occurred, check the dinfo and the niters arguments for more details.

CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED  
The configuration parameter gesv_irs_params structure was not created.

CUSOLVER_STATUS_IRS_PARAMS_INVALID  
One of the configuration parameter in the gesv_irs_params structure is not valid.

CUSOLVER_STATUS_IRS_PARAMS_INVALID_PREC  
The main and/or the lowest precision configuration parameter in the gesv_irs_params structure is not valid, check the table above for the supported combinations.

CUSOLVER_STATUS_IRS_PARAMS_INVALID_MAXITER  
The maxiter configuration parameter in the gesv_irs_params structure is not valid.

CUSOLVER_STATUS_IRS_PARAMS_INVALID_REFINE  
The refinement solver configuration parameter in the gesv_irs_params structure is not valid.

CUSOLVER_STATUS_IRS_NOT_SUPPORTED  
One of the configuration parameter in the gesv_irs_params structure is not supported. For example if nrhs >1, and refinement solver was set to CUSOLVER_IRS_REFINE_GMRES.

CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED  
The information structure gesv_irs_infos was not created.

CUSOLVER_STATUS_ALLOC_FAILED  
CPU memory allocation failed, most likely during the allocation of the residual array that store the residual norms.

2.4.2.12. cusolverDn<t>geqrf()
These helper functions calculate the size of work buffers needed.

```c
 cusolverStatus_t cusolverDnSgeqrf_bufferSize(cusolverDnHandle_t handle, 
                                           int m, 
                                           int n, 
                                           float *A, 
                                           int lda, 
                                           int *Lwork );
```

```c
 cusolverStatus_t cusolverDnDgeqrf_bufferSize(cusolverDnHandle_t handle, 
```

```c```
Using the CUSOLVER API

int m,
int n,
double *A,
int lda,
int *Lwork );

cusolverStatus_t
cusolverDnCgeqrf_bufferSize(cusolverDnHandle_t handle,
int m,
int n,
cuComplex *A,
int lda,
int *Lwork );

cusolverStatus_t
cusolverDnZgeqrf_bufferSize(cusolverDnHandle_t handle,
int m,
int n,
cuDoubleComplex *A,
int lda,
int *Lwork );

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSgeqrf(cusolverDnHandle_t handle,
int m,
int n,
float *A,
int lda,
float *TAU,
float *Workspace,
int Lwork,
int *devInfo );

cusolverStatus_t
cusolverDnDgeqrf(cusolverDnHandle_t handle,
int m,
int n,
double *A,
int lda,
double *TAU,
double *Workspace,
int Lwork,
int *devInfo );

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnCgeqrf(cusolverDnHandle_t handle,
int m,
int n,
cuComplex *A,
int lda,
cuComplex *TAU,
cuComplex *Workspace,
int Lwork,
int *devInfo );

cusolverStatus_t
cusolverDnZgeqrf(cusolverDnHandle_t handle,
This function computes the QR factorization of a \( m \times n \) matrix

\[
A = Q^* R
\]

where \( A \) is a \( m \times n \) matrix, \( Q \) is a \( m \times n \) matrix, and \( R \) is a \( n \times n \) upper triangular matrix.

The user has to provide working space which is pointed by input parameter workspace. The input parameter Lwork is size of the working space, and it is returned by geqrf_bufferSize().

The matrix \( R \) is overwritten in upper triangular part of \( A \), including diagonal elements.

The matrix \( Q \) is not formed explicitly, instead, a sequence of householder vectors are stored in lower triangular part of \( A \). The leading nonzero element of householder vector is assumed to be 1 such that output parameter \( TAU \) contains the scaling factor \( \tau \). If \( v \) is original householder vector, \( q \) is the new householder vector corresponding to \( \tau \), satisfying the following relation

\[
I - 2v v^H = I - \tau q q^H
\]

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle).

**API of geqrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>( \langle \text{type} \rangle ) array of dimension ( l\text{da} \times n ) with ( l\text{da} ) is not less than ( \max(1,m) ).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>TAU</td>
<td>device</td>
<td>output</td>
<td>( \langle \text{type} \rangle ) array of dimension at least ( \min(m,n) ).</td>
</tr>
<tr>
<td>Workspace</td>
<td>device</td>
<td>in/out</td>
<td>working space, ( \langle \text{type} \rangle ) array of size ( \text{Lwork} ).</td>
</tr>
<tr>
<td>Lwork</td>
<td>host</td>
<td>input</td>
<td>size of working array Workspace.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the LU factorization is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully. |
CUSOLVER_API

The library was not initialized.

CUSOLVER_STATUS_INVALID_VALUE

invalid parameters were passed \(m, n \leq 0\) or \(lda < \max(1, m)\).

CUSOLVER_STATUS_ARCH_MISMATCH

the device only supports compute capability 2.0 and above.

CUSOLVER_STATUS_INTERNAL_ERROR

an internal operation failed.

### 2.4.2.13. cusolverDnGeqrf()

[[DEPRECATED]] use cusolverDnXgeqrf() instead. The routine will be removed in the next major release.

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverDnGeqrf_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeTau,
    const void *tau,
    cudaDataType computeType,
    size_t *workspaceInBytes )
```

The routine bellow

```c
cusolverStatus_t
cusolverDnGeqrf(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType dataTypeTau,
    void *tau,
    cudaDataType computeType,
    void *pBuffer,
    size_t workspaceInBytes,
    int *info )
```

computes the QR factorization of a \(m \times n\) matrix

\[ A = Q \cdot R \]

where \(A\) is a \(m \times n\) matrix, \(Q\) is a \(m \times n\) matrix, and \(R\) is a \(n \times n\) upper triangular matrix using the generic API interface.

The user has to provide working space which is pointed by input parameter \(pBuffer\). The input parameter \(workspaceInBytes\) is size in bytes of the working space, and it is returned by cusolverDnGeqrf_bufferSize().
The matrix $R$ is overwritten in upper triangular part of $A$, including diagonal elements.

The matrix $Q$ is not formed explicitly, instead, a sequence of householder vectors are stored in lower triangular part of $A$. The leading nonzero element of householder vector is assumed to be $1$ such that output parameter $\tau$ contains the scaling factor $\tau$. If $v$ is original householder vector, $q$ is the new householder vector corresponding to $\tau$, satisfying the following relation

$$ I - 2vv^H = I - \tauqq^H $$

If output parameter $\text{info} = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle).

Currently, 

**Table of algorithms supported by cusolverDnGeqrf**

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for cusolverDnGeqrf_bufferSize and cusolverDnGeqrf:

**API of geqrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension $lda \times n$ with $lda$ is not less than $\max(1,m)$.</td>
</tr>
<tr>
<td>$lda$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>device</td>
<td>output</td>
<td>array of dimension at least $\min(m,n)$.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>in/out</td>
<td>Working space. Array of type void of size workspaceInBytes bytes.</td>
</tr>
<tr>
<td>workspaceInBytes</td>
<td>host</td>
<td>input</td>
<td>size in bytes of working array pBuffer.</td>
</tr>
<tr>
<td>$\text{info}$</td>
<td>device</td>
<td>output</td>
<td>if $\text{info} = 0$, the LU factorization is successful. if $\text{info} = -i$, the $i$-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, $\text{dataTypeA}$ is data type of the matrix $A$ and array $\tau$ and $\text{computeType}$ is compute type of the operation. cusolverDnGeqrf only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>$\text{DataTypeA}$</th>
<th>$\text{ComputeType}$</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGEQRF</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

2.4.2.14. cusolverDn<t1><t2>gels()

These functions compute the solution of a system of linear equations with one or multiple right hand sides using mixed precision iterative refinement techniques based on the QR factorization Xgels. These functions are similar in term of functionalities to the full precision LAPACK QR (least squares) solver (Xgels, where X denotes Z,C,D,S) but it uses lower precision internally in order to provide faster time to solution, from here comes the name mixed precision. Mixed precision iterative refinement techniques means that the solver compute an QR factorization in lower precision and then iteratively refine the solution to achieve the accuracy of the Inputs/Outputs datatype precision. The <t1> corresponds to the Inputs/Outputs datatype precision while <t2> represent the internal lower precision at which the factorization will be carried on.

\[ A \times X = B \]

Where \( A \) is \( m \)-by-\( n \) matrix and \( X \) is \( n \)-by-\( nrhs \) and \( B \) is \( m \)-by-\( nrhs \) matrices.

Functions API are designed to be as close as possible to LAPACK API to be considered as a quick and easy drop-in replacement. Description of these functions is given below.

\(<t1><t2>gels()\) functions are designated by two floating point precisions The \(<t1>\) corresponds to the main precision (e.g., Inputs/Outputs datatype precision) and the \(<t2>\) represent the internal lower precision at which the factorization will be carried on. cusolver\(<t1><t2>gels()\) first attempts to factorize the matrix in lower precision and use this factorization within an iterative refinement procedure to obtain a solution with same normwise backward error as the main precision \(<t1>\). If the approach fails to converge, then the method fallback to the main precision factorization and solve \(Xgels\) such a way that there is always a good solution at the output of these functions. If \(<t2>\) is equal to \(<t1>\), then it is not a mixed precision process but rather a full one precision factorisation, solve and refinement within the same main precision.

The iterative refinement process is stopped if

\[ \text{ITER} > \text{ITERMAX} \]

or for all the RHS we have:

\[ \text{RNRM} < \sqrt{N} \times \text{XNRM} \times \text{ANRM} \times \text{EPS} \times \text{BWDMAX} \]
where

- ITER is the number of the current iteration in the iterative refinement process
- RNRM is the infinity-norm of the residual
- XNRM is the infinity-norm of the solution
- ANRM is the infinity-operator-norm of the matrix A
- EPS is the machine epsilon that matches LAPACK `<t1>LAMCH('Epsilon')`

The value ITERMAX and BWDMAX are fixed to 50 and 1.0 respectively.

The function returns value describes the results of the solving process. A CUSOLVER_STATUS_SUCCESS indicates that the function finished with success otherwise, it indicates if one of the API arguments is incorrect, or if the function did not finish with success. More details about the error will be in the `niter` and the `dinfo` API parameters. See their description below for more details. User should provide the required workspace allocated on device memory. The amount of bytes required can be queried by calling the respective function `<t1>`.<t2>`gels_bufferSize()`.

We provide a large set of mixed precision functions that include half, bfloat and tensorfloat as a lower precision as well as same precision functions (e.g., main and lowest precision are equal `<t2>` is equal to `<t1>`). The following table specifies which precisions will be used for which interface function:

Tensor Float (TF32), introduced with NVIDIA Ampere Architecture GPUs, is the most robust tensor core accelerated compute mode for the iterative refinement solver. It is able to solve the widest range of problems in HPC arising from different applications and provides up to 4X and 5X speedup for real and complex systems, respectively. On Volta and Turing architecture GPUs, half precision tensor core acceleration is recommended. In cases where the iterative refinement solver fails to converge to the desired accuracy (main precision, INOUT data precision), it is recommended to use main precision as internal lowest precision (i.e., `cusolverDn[DD,ZZ]gels` for the FP64 case).

### Supported combinations of floating point precisions for cusolver `<t1>`.<t2>`gels()` functions

<table>
<thead>
<tr>
<th>Interface function</th>
<th>Main precision (matrix, rhs and solution datatype)</th>
<th>Lowest precision allowed to be used internally</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cusolverDnZZgels</code></td>
<td><code>cuDoubleComplex</code></td>
<td><code>double complex</code></td>
</tr>
<tr>
<td><code>cusolverDnZCgels</code></td>
<td><code>cuDoubleComplex</code></td>
<td><code>single complex</code></td>
</tr>
<tr>
<td><code>cusolverDnZKgels</code></td>
<td><code>cuDoubleComplex</code></td>
<td><code>half complex</code></td>
</tr>
<tr>
<td><code>cusolverDnZEgels</code></td>
<td><code>cuDoubleComplex</code></td>
<td><code>bfloat complex</code></td>
</tr>
<tr>
<td><code>cusolverDnZYgels</code></td>
<td><code>cuDoubleComplex</code></td>
<td><code>tensorfloat complex</code></td>
</tr>
<tr>
<td><code>cusolverDnCCgels</code></td>
<td><code>cuComplex</code></td>
<td><code>single complex</code></td>
</tr>
<tr>
<td><code>cusolverDnCKgels</code></td>
<td><code>cuComplex</code></td>
<td><code>half complex</code></td>
</tr>
<tr>
<td><code>cusolverDnCEgels</code></td>
<td><code>cuComplex</code></td>
<td><code>bfloat complex</code></td>
</tr>
<tr>
<td><code>cusolverDnCYgels</code></td>
<td><code>cuComplex</code></td>
<td><code>tensorfloat complex</code></td>
</tr>
<tr>
<td><code>cusolverDnDDgels</code></td>
<td><code>double</code></td>
<td><code>double</code></td>
</tr>
</tbody>
</table>
### cusolverDn<t1><t2>gels_bufferSize() Functions

The `cusolverDn<t1><t2>gels_bufferSize()` functions will return workspace buffer size in bytes required for the corresponding `cusolverDn<t1><t2>gels()` function.

#### cusolverStatus_t

**cusolverDnZZgels_bufferSize**

```c
cusolverStatus_t
cusolverDnZZgels_bufferSize(
    cusolverHandle_t handle,
    int m,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int lda,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);
```

**cusolverDnZCgels_bufferSize**

```c
cusolverStatus_t
cusolverDnZCgels_bufferSize(
    cusolverHandle_t handle,
    int m,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int lda,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);
```

**cusolverDnZKgels_bufferSize**

```c
cusolverStatus_t
cusolverDnZKgels_bufferSize(
    cusolverHandle_t handle,
    int m,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int lda,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dwork,
    size_t * lwork_bytes);
```
Using the CUSOLVER API

cusolverStatus_t
cusolverDnZEgels_bufferSize(
    cusolverHandle_t    handle,
    int               m,
    int               n,
    int               nrhs,
    cuDoubleComplex   *   dA,
    int               ldaA,
    cuDoubleComplex   *   dB,
    int               ldbB,
    cuDoubleComplex   *   dX,
    int               lddX,
    void              *   dwork,
    size_t            *   lwork_bytes);}

cusolverStatus_t
cusolverDnZYgels_bufferSize(
    cusolverHandle_t    handle,
    int               m,
    int               n,
    int               nrhs,
    cuDoubleComplex   *   dA,
    int               ldaA,
    cuDoubleComplex   *   dB,
    int               ldbB,
    cuDoubleComplex   *   dX,
    int               lddX,
    void              *   dwork,
    size_t            *   lwork_bytes);}

cusolverStatus_t
cusolverDnCCgels_bufferSize(
    cusolverHandle_t    handle,
    int               m,
    int               n,
    int               nrhs,
    cuComplex         *   dA,
    int               ldaA,
    cuComplex         *   dB,
    int               ldbB,
    cuComplex         *   dX,
    int               lddX,
    void              *   dwork,
    size_t            *   lwork_bytes);}

cusolverStatus_t
cusolverDnCKgels_bufferSize(
    cusolverHandle_t    handle,
    int               m,
    int               n,
    int               nrhs,
    cuComplex         *   dA,
    int               ldaA,
    cuComplex         *   dB,
    int               ldbB,
    cuComplex         *   dX,
    int               lddX,
    void              *   dwork,
    size_t            *   lwork_bytes);
cusolverStatus_t
cusolverDnCEgels_bufferSize(
cusolverHandle_t                handle,
int                             m,
int                             n,
int                             nrhs,
cuComplex                   *   dA,
int                             ldda,
cuComplex                   *   dB,
int                             lddb,
cuComplex                   *   dX,
int                             lddx,
void                        *   dwork,
size_t                      *   lwork_bytes);

cusolverStatus_t
cusolverDnCYgels_bufferSize(
cusolverHandle_t                handle,
int                             m,
int                             n,
int                             nrhs,
cuComplex                   *   dA,
int                             ldda,
cuComplex                   *   dB,
int                             lddb,
cuComplex                   *   dX,
int                             lddx,
void                        *   dwork,
size_t                      *   lwork_bytes);

cusolverStatus_t
cusolverDnDDgels_bufferSize(
cusolverHandle_t                handle,
int                             m,
int                             n,
int                             nrhs,
double                      *   dA,
int                             ldda,
double                      *   dB,
int                             lddb,
double                      *   dX,
int                             lddx,
void                        *   dwork,
size_t                      *   lwork_bytes);

cusolverStatus_t
cusolverDnDSgels_bufferSize(
cusolverHandle_t                handle,
int                             m,
int                             n,
int                             nrhs,
double                      *   dA,
int                             ldda,
double                      *   dB,
int                             lddb,
double                      *   dX,
int                             lddx,
void                        *   dwork,
size_t                      *   lwork_bytes);
**CUSOLVER API**

**cuSOLVER Library**

```c
/* cuSOLVER DnDHgels_bufferSize */
cusolverStatus_t
cusolverDnDHgels_bufferSize(
   cusolverHandle_t       handle,
   int                    m,
   int                    n,
   int                    nrhs,
   double                 *  dA,
   int                    ldaA,
   double                 *  dB,
   int                    ldbB,
   double                 *  dX,
   int                    ldx,
   void                   *  dwork,
   size_t                 *  lwork_bytes);

/* cuSOLVER DnDBgels_bufferSize */
cusolverStatus_t
 cusolverDnDBgels_bufferSize(
   cusolverHandle_t       handle,
   int                    m,
   int                    n,
   int                    nrhs,
   double                 *  dA,
   int                    ldaA,
   double                 *  dB,
   int                    ldbB,
   double                 *  dX,
   int                    ldx,
   void                   *  dwork,
   size_t                 *  lwork_bytes);

/* cuSOLVER DnDXgels_bufferSize */
cusolverStatus_t
 cusolverDnDXgels_bufferSize(
   cusolverHandle_t       handle,
   int                    m,
   int                    n,
   int                    nrhs,
   double                 *  dA,
   int                    ldaA,
   double                 *  dB,
   int                    ldbB,
   double                 *  dX,
   int                    ldx,
   void                   *  dwork,
   size_t                 *  lwork_bytes);

/* cuSOLVER DnSSgels_bufferSize */
cusolverStatus_t
 cusolverDnSSgels_bufferSize(
   cusolverHandle_t       handle,
   int                    m,
   int                    n,
   int                    nrhs,
   float                  *  dA,
   int                    ldaA,
   float                  *  dB,
   int                    ldbB,
   float                  *  dX,
   int                    ldx,
   void                   *  dwork,
   size_t                 *  lwork_bytes);

/* cuSOLVER DnSHgels_bufferSize */
cusolverStatus_t
 cusolverDnSHgels_bufferSize(
   cusolverHandle_t       handle,
   int                    m,
   int                    n,
   int                    nrhs,
   double                 *  dA,
   int                    ldaA,
   double                 *  dB,
   int                    ldbB,
   double                 *  dX,
   int                    ldx,
   void                   *  dwork,
   size_t                 *  lwork_bytes);
```

**cuSOLVER Library**
Using the CUSOLVER API

```c
// cusolverDnSBgels_bufferSize() function

// cusolverDnSXgels_bufferSize() function

cusolverStatus_t
 cusolverDnSBgels_bufferSize(
    cusolverHandle_t           handle,
    int                        m,
    int                        n,
    int                        nrhs,
    float*                    dA,
    int                        ldda,
    float*                    dB,
    int                        lddb,
    float*                    dX,
    int                        lddx,
    void*                     dwork,
    size_t*                   lwork_bytes);

cusolverStatus_t
 cusolverDnSXgels_bufferSize(
    cusolverHandle_t           handle,
    int                        m,
    int                        n,
    int                        nrhs,
    float*                    dA,
    int                        ldda,
    float*                    dB,
    int                        lddb,
    float*                    dX,
    int                        lddx,
    void*                     dwork,
    size_t*                   lwork_bytes);

Parameters of cusolverDn<T1><T2>gels_bufferSize() functions

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDN library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of rows of the matrix A. Should be non-negative and n&lt;=m.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of columns of the matrix A. Should be non-negative and n&lt;=m.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>None</td>
<td>Matrix A with size m-by-n. Can be NULL.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of two-dimensional array used to store matrix A. ldda &gt;= m.</td>
</tr>
</tbody>
</table>
```
### Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dB</td>
<td>device</td>
<td>None</td>
<td>Set of right hand sides B of size m-by-nrhs. Can be NULL.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of right hand sides B. lddb &gt;= max(1,m,n).</td>
</tr>
<tr>
<td>dX</td>
<td>device</td>
<td>None</td>
<td>Set of solution vectors X of size n-by-nrhs. Can be NULL.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of solution vectors X. lddx &gt;= max(1,m,n).</td>
</tr>
<tr>
<td>dwork</td>
<td>device</td>
<td>none</td>
<td>Pointer to device workspace. Not used and can be NULL.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>output</td>
<td>Pointer to a variable where required size of temporary workspace in bytes will be stored. Can’t be NULL.</td>
</tr>
</tbody>
</table>

```c

cusolverStatus_t cusolverDnZZgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    cuDoubleComplex     *   dA,
    int                     ldda,
    cuDoubleComplex     *   dB,
    int                     lddb,
    cuDoubleComplex     *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);

cusolverStatus_t cusolverDnZCgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    cuDoubleComplex     *   dA,
    int                     ldda,
    cuDoubleComplex     *   dB,
    int                     lddb,
    cuDoubleComplex     *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);

cusolverStatus_t cusolverDnZKgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    cuDoubleComplex     *   dA,
    int                     ldda,
    cuDoubleComplex     *   dB,
    int                     lddb,
    cuDoubleComplex     *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);
```
Using the CUSOLVER API

```c
int lddb,
cuDoubleComplex * dX,
int lddx,
void * dWorkspace,
size_t lwork_bytes,
int * niter,
int * dinfo);

cusolverStatus_t cusolverDnZEgels(
    cusolverDnHandle_t      handle,
    int m,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int ldda,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnZYgels(
    cusolverDnHandle_t      handle,
    int m,
    int n,
    int nrhs,
    cuDoubleComplex * dA,
    int ldda,
    cuDoubleComplex * dB,
    int lddb,
    cuDoubleComplex * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnCCgels(
    cusolverDnHandle_t      handle,
    int m,
    int n,
    int nrhs,
    cuComplex * dA,
    int ldda,
    cuComplex * dB,
    int lddb,
    cuComplex * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);

cusolverStatus_t cusolverDnCKgels(
    cusolverDnHandle_t      handle,
    int m,
    int n,
    int nrhs,
    cuComplex * dA,
    int ldda,
    cuComplex * dB,
    int lddb,
    cuComplex * dX,
    int lddx,
    void * dWorkspace,
    size_t lwork_bytes,
    int * niter,
    int * dinfo);
```
using the cusolver api

```c
#include <cusolverDn.h>

cusolverStatus_t cusolverDnCegels(cusolverDnHandle_t handle, int m, int n, int nrhs, cuComplex * dA, int ldda, cuComplex * dB, int lddb, cuComplex * dX, int lddx, void * dWorkspace, size_t lwork_bytes, int * niter, int * dinfo);

cusolverStatus_t cusolverDnCygels(cusolverDnHandle_t handle, int m, int n, int nrhs, cuComplex * dA, int ldda, cuComplex * dB, int lddb, cuComplex * dX, int lddx, void * dWorkspace, size_t lwork_bytes, int * niter, int * dinfo);

cusolverStatus_t cusolverDnDdgels(cusolverDnHandle_t handle, int m, int n, int nrhs, double * dA, int ldda, double * dB, int lddb, double * dX, int lddx, void * dWorkspace, size_t lwork_bytes, int * niter, int * dinfo);

cusolverStatus_t cusolverDnDsgegs(cusolverDnHandle_t handle, int m, int n, int nrhs, double * dA, int ldda, double * dB, int lddb, double * dX, int lddx, void * dWorkspace, size_t lwork_bytes, int * niter, int * dinfo);
```

CuSOLVER Library
Using the CUSOLVER API

cusolverStatus_t cusolverDnDHgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    double              *   dA,
    int                     ldda,
    double              *   dB,
    int                     lddb,
    double              *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);

cusolverStatus_t cusolverDnDBgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    double              *   dA,
    int                     ldda,
    double              *   dB,
    int                     lddb,
    double              *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);

cusolverStatus_t cusolverDnDXgels(
    cusolverDnHandle_t      handle,
    int                     m,
    int                     n,
    int                     nrhs,
    double              *   dA,
    int                     ldda,
    double              *   dB,
    int                     lddb,
    double              *   dX,
    int                     lddx,
    void                *   dWorkspace,
    size_t                  lwork_bytes,
    int                     *   niter,
    int                     *   dinfo);
CUSOLVER API Functions

```c
 cusolverStatus_t cusolverDnSSgels(
  cusolverDnHandle_t      handle,
  int                     m,
  int                     n,
  int                     nrhs,
  float               *   dA,
  int                     ldda,
  float               *   dB,
  int                     lddb,
  float               *   dX,
  int                     lddx,
  void                *   dWorkspace,
  size_t                  lwork_bytes,
  int*                    niter,
  int*                    dinfo);

 cusolverStatus_t cusolverDnSHgels(
  cusolverDnHandle_t      handle,
  int                     m,
  int                     n,
  int                     nrhs,
  float               *   dA,
  int                     ldda,
  float               *   dB,
  int                     lddb,
  float               *   dX,
  int                     lddx,
  void                *   dWorkspace,
  size_t                  lwork_bytes,
  int*                    niter,
  int*                    dinfo);

 cusolverStatus_t cusolverDnSBgels(
  cusolverDnHandle_t      handle,
  int                     m,
  int                     n,
  int                     nrhs,
  float               *   dA,
  int                     ldda,
  float               *   dB,
  int                     lddb,
  float               *   dX,
  int                     lddx,
  void                *   dWorkspace,
  size_t                  lwork_bytes,
  int*                    niter,
  int*                    dinfo);

 cusolverStatus_t cusolverDnSXgels(
  cusolverDnHandle_t      handle,
  int                     m,
  int                     n,
  int                     nrhs,
  float               *   dA,
  int                     ldda,
  float               *   dB,
  int                     lddb,
  float               *   dX,
  int                     lddx,
  void                *   dWorkspace,
  size_t                  lwork_bytes,
  int*                    niter,
  int*                    dinfo);
```

Using the CUSOLVER API

The CUSOLVER API functions for solving linear systems with general matrices are as follows:

- `cusolverDnSSgels` for solving a system with a general matrix using a square matrix.
- `cusolverDnSHgels` for solving a system with a general matrix using a symmetric matrix.
- `cusolverDnSBgels` for solving a system with a general matrix using a banded matrix.
- `cusolverDnSXgels` for solving a system with a general matrix using a sparse matrix.

For more details, refer to the CUSOLVER documentation.
Using the CUSOLVER API

```c
size_t lwork_bytes,
int * niter,
int * dinfo);
```

### Parameters of cusolverDn<T1><T2>gels() functions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDN library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of rows of the matrix A. Should be non-negative and n&lt;=m</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of columns of the matrix A. Should be non-negative and n&lt;=m</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>in/out</td>
<td>Matrix A with size m-by-n. Can’t be NULL. On return - unchanged if the lowest precision is not equal to the main precision and the iterative refinement solver converged, - garbage otherwise.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. ldda &gt;= m.</td>
</tr>
<tr>
<td>dB</td>
<td>device</td>
<td>input</td>
<td>Set of right hand sides B of size m-by-nrhs. Can’t be NULL.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of right hand sides B. lddb &gt;= max(1,m,n).</td>
</tr>
<tr>
<td>dX</td>
<td>device</td>
<td>output</td>
<td>Set of solution vectors X of size n-by-nrhs. Can’t be NULL.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of solution vectors X. lddx &gt;= max(1,m,n).</td>
</tr>
<tr>
<td>dWorkspace</td>
<td>device</td>
<td>input</td>
<td>Pointer to an allocated workspace in device memory of size lwork_bytes.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>input</td>
<td>Size of the allocated device workspace. Should be at least what was returned by cusolverDn&lt;T1&gt;&lt;T2&gt;gels_bufferSize() function</td>
</tr>
<tr>
<td>niter</td>
<td>host</td>
<td>output</td>
<td>If iter is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-0 : iterative refinement has failed, main precision [Inputs/Outputs precision] factorization has been performed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1 : taking into account machine parameters, n, nrhs, it is a priori not worth working in lower precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2 : overflow of an entry when moving from main to lower precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3 : failure during the factorization</td>
</tr>
</tbody>
</table>
-5 : overflow occurred during computation
-50: solver stopped the iterative refinement after reaching maximum allowed iterations.
>0 : iter is a number of iterations solver performed to reach convergence criteria

<table>
<thead>
<tr>
<th>dinfo</th>
<th>device</th>
<th>output</th>
<th>Status of the IRS solver on the return. If 0 - solve was successful. If dinfo = -i then i-th argument is not valid.</th>
</tr>
</thead>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed, for example:</td>
</tr>
<tr>
<td></td>
<td>▶ n&lt;0</td>
</tr>
<tr>
<td></td>
<td>▶ ldda&lt;max(1,m)</td>
</tr>
<tr>
<td></td>
<td>▶ ldde&lt;max(1,m,n)</td>
</tr>
<tr>
<td></td>
<td>▶ ldde&lt;max(1,m,n)</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the IRS solver supports compute capability 7.0 and above. The lowest precision options CUSOLVER_[CR]<em>16BF and CUSOLVER</em>[CR]_TF32 are only available on compute capability 8.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_WORKSPACE</td>
<td>lwork_bytes is smaller than the required workspace.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_OUT_OF_RANGE</td>
<td>Numerical error related to niters &lt;0, see niters description for more details.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal error occurred, check the dinfo and the niters arguments for more details.</td>
</tr>
</tbody>
</table>

### 2.4.2.15. cusolverDnIRSXgels()

This function is designed to perform same functionality as cusolverDn<T1><T2>gels() functions, but wrapped in a more generic and expert interface that gives user more control to parametrize the function as well as it provides more informations on output. cusolverDnIRSXgels() allows additional control of the solver parameters such as setting:

- the main precision (Inputs/Outputs precision) of the solver,
- the lowest precision to be used internally by the solver,
- the refinement solver type,
- the maximum allowed number of iterations in the refinement phase,
- the tolerance of the refinement solver,
Using the CUSOLVER API

- the fallback to main precision,
- and more other.

through the configuration parameters structure gels_irs_params and its helper functions. For more details about what configuration can be set and its meaning please refer to all the functions in the cuSolverDN Helper Function Section that start with cusolverDnIRSParmsxxxx(). Moreover, cusolverDnIRSxgels() provides additional informations on the output such as the convergence history (e.g., the residual norms) at each iteration and the number of iterations needed to converge. For more details about what informations can be retrieved and its meaning please refer to all the functions in the cuSolverDN Helper Function Section that start with cusolverDnIRSInfosxxxx().

The function returns value describes the results of the solving process. A CUSOLVER_STATUS_SUCCESS indicates that the function finished with success otherwise, it indicates if one of the API arguments is incorrect, or if the configurations of params/infos structure is incorrect or if the function did not finish with success. More details about the error can be found by checking the nитers and the dinfo API parameters. See their description below for further details. User should provide the required workspace allocated on device for the cusolverDnIRSxgels() function. The amount of bytes required for the function can be queried by calling the respective function cusolverDnIRSxgels_bufferSize(). Note that, if the user would like a particular configuration to be set via the params structure, it should be set before the call to cusolverDnIRSxgels_bufferSize() to get the size of the required workspace.

The following table provides all possible combinations values for the lowest precision corresponding to the Inputs/Outputs data type. Note that if the lowest precision matches the Inputs/Outputs datatype, then main precision factorization will be used.

Tensor Float (TF32), introduced with NVIDIA Ampere Architecture GPUs, is the most robust tensor core accelerated compute mode for the iterative refinement solver. It is able to solve the widest range of problems in HPC arising from different applications and provides up to 4X and 5X speedup for real and complex systems, respectively. On Volta and Turing architecture GPUs, half precision tensor core acceleration is recommended. In cases where the iterative refinement solver fails to converge to the desired accuracy (main precision, INOUT data precision), it is recommended to use main precision as internal lowest precision.

**Supported Inputs/Outputs data type and lower precision for the IRS solver**

<table>
<thead>
<tr>
<th>Inputs/Outputs Data Type (e.g., main precision)</th>
<th>Supported values for the lowest precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_C_64F</td>
<td>CUSOLVER_C_64F, CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32</td>
</tr>
<tr>
<td>CUSOLVER_C_32F</td>
<td>CUSOLVER_C_32F, CUSOLVER_C_16F, CUSOLVER_C_16BF, CUSOLVER_C_TF32</td>
</tr>
<tr>
<td>CUSOLVER_R_64F</td>
<td>CUSOLVER_R_64F, CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32</td>
</tr>
<tr>
<td>CUSOLVER_R_32F</td>
<td>CUSOLVER_R_32F, CUSOLVER_R_16F, CUSOLVER_R_16BF, CUSOLVER_R_TF32</td>
</tr>
</tbody>
</table>

The cusolverDnIRSxgels_bufferSize() function return the required workspace buffer size in bytes for the corresponding cusolverDnXgels() call with given gels_irs_params configuration.
### Parameters of cusolverDnIRSxgels_bufferSize() functions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDn library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>Xgels configuration parameters</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of rows of the matrix $A$. Should be non-negative and $n &lt;= m$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of columns of the matrix $A$. Should be non-negative and $n &lt;= m$.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative. Note that, nrhs is limited to 1 if the selected IRS refinement solver is CUSOLVER_IRS_REFINE_GMRES, CUSOLVER_IRS_REFINE_GMRES_GMRES, CUSOLVER_IRS_REFINE_CLASSICAL_GMRES.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>out</td>
<td>Pointer to a variable, where the required size in bytes, of the workspace will be stored after a call to cusolverDnIRSxgels_bufferSize. Can't be NULL.</td>
</tr>
</tbody>
</table>

### Parameters of cusolverDnIRSxgels() functions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cusolverDn library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>Xgels configuration parameters</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of rows of the matrix $A$. Should be non-negative and $n &lt;= m$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of columns of the matrix $A$. Should be non-negative and $n &lt;= m$.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative. Note that, nrhs is limited to 1 if the selected IRS refinement solver is CUSOLVER_IRS_REFINE_GMRES, CUSOLVER_IRS_REFINE_GMRES_GMRES, CUSOLVER_IRS_REFINE_CLASSICAL_GMRES.</td>
</tr>
<tr>
<td>dA</td>
<td>host</td>
<td>input</td>
<td>Matrix $A$.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of matrix $A$.</td>
</tr>
<tr>
<td>dB</td>
<td>host</td>
<td>input</td>
<td>Matrix $B$.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of matrix $B$.</td>
</tr>
<tr>
<td>dX</td>
<td>host</td>
<td>input</td>
<td>Matrix $X$.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of matrix $X$.</td>
</tr>
<tr>
<td>dWorkspace</td>
<td>host</td>
<td>input</td>
<td>Workspace.</td>
</tr>
<tr>
<td>dinfo</td>
<td>host</td>
<td>output</td>
<td>Information from solver.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gels_irs_params</td>
<td>host</td>
<td>input</td>
<td>Configuration parameters structure, can serve one or more calls to any IRS solver</td>
</tr>
<tr>
<td>gels_irs_infos</td>
<td>host</td>
<td>in/out</td>
<td>Info structure, where information about a particular solve will be stored. The gels_irs_infos structure correspond to a particular call. Thus different calls requires different gels_irs_infos structure otherwise, it will be overwritten.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of rows of the matrix $A$. Should be non-negative and $n \leq m$</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of columns of the matrix $A$. Should be non-negative and $n \leq m$.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>Number of right hand sides to solve. Should be non-negative. Note that, $nrhs$ is limited to 1 if the selected IRS refinement solver is CUSOLVER IRS_REFINE_GMRES, CUSOLVER IRS_REFINE_GMRES_GMRES, CUSOLVER IRS_REFINE_CLASSICAL_GMRES.</td>
</tr>
<tr>
<td>dA</td>
<td>device</td>
<td>in/out</td>
<td>Matrix $A$ with size $m \times n$. Can’t be NULL. On return - unchanged if the lowest precision is not equal to the main precision and the iterative refinement solver converged, - garbage otherwise.</td>
</tr>
<tr>
<td>ldda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$. $ldda \geq m$.</td>
</tr>
<tr>
<td>dB</td>
<td>device</td>
<td>input</td>
<td>Set of right hand sides $B$ of size $m \times nrhs$. Can’t be NULL.</td>
</tr>
<tr>
<td>lddb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of right hand sides $B$. $lddb \geq \max(1,m,n)$.</td>
</tr>
<tr>
<td>dX</td>
<td>device</td>
<td>output</td>
<td>Set of solution vectors $X$ of size $n \times nrhs$. Can’t be NULL.</td>
</tr>
<tr>
<td>lddx</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix of solution vectors $X$. $lddx \geq \max(1,m,n)$.</td>
</tr>
<tr>
<td>dWorkspace</td>
<td>device</td>
<td>input</td>
<td>Pointer to an allocated workspace in device memory of size lwork_bytes.</td>
</tr>
<tr>
<td>lwork_bytes</td>
<td>host</td>
<td>input</td>
<td>Size of the allocated device workspace. Should be at least what was returned by cusolverDnIRSXgels_bufferSize() function</td>
</tr>
<tr>
<td>niters</td>
<td>host</td>
<td>output</td>
<td>If iter is $\leq 0$ : iterative refinement has failed, main precision (Inputs/Outputs precision) factorization has been performed if fallback is enabled.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

-1 : taking into account machine parameters, n, nrhs, it is a priori not worth working in lower precision
-2 : overflow of an entry when moving from main to lower precision
-3 : failure during the factorization
-5 : overflow occured during computation
-maxiter: solver stopped the iterative refinement after reaching maximum allowed iterations.
>0 : iter is a number of iterations solver performed to reach convergence criteria

dinfo device output
Status of the IRS solver on the return. If 0 - solve was successful. If dinfo = -i then i-th argument is not valid.

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed, for example:</td>
</tr>
<tr>
<td></td>
<td>▶ n&lt;0</td>
</tr>
<tr>
<td></td>
<td>▶ ldda&lt;max(1,m)</td>
</tr>
<tr>
<td></td>
<td>▶ lddb&lt;max(1,m,n)</td>
</tr>
<tr>
<td></td>
<td>▶ lddx&lt;max(1,m,n)</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the IRS solver supports compute capability 7.0 and above. The lowest precision options CUSOLVER_[CR]<em>16BF and CUSOLVER</em>[CR]_TF32 are only available on compute capability 8.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_WORKSPACE</td>
<td>lwork_bytes is smaller than the required workspace. Could happen if the users called cusolverDnIRSXgels_bufferSize() function, then changed some of the configurations setting such as the lowest precision.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_OUT_OF_RANGE</td>
<td>Numerical error related to niters &lt;0, see niters description for more details.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal error occured, check the dinfo and the niters arguments for more details.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_NOT_INITIALIZED</td>
<td>The configuration parameter gels_irs_params structure was not created.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_IRS_PARAMS_INVALID</td>
<td>One of the configuration parameter in the gels_irs_params structure is not valid.</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_IRS_PARAMS_INVALID_PREC | The main and/or the lowest precision configuration parameter in the gels_irs_params structure is not valid, check the table above for the supported combinations.

CUSOLVER_STATUS_IRS_PARAMS_INVALID_MAXITER | The maxiter configuration parameter in the gels_irs_params structure is not valid.

CUSOLVER_STATUS_IRS_PARAMS_INVALID_REFINE | The refinement solver configuration parameter in the gels_irs_params structure is not valid.

CUSOLVER_STATUS_IRS_NOT_SUPPORTED | One of the configuration parameter in the gels_irs_params structure is not supported. For example if nrhs >1, and refinement solver was set to CUSOLVER_IRS_REFINE_GMRES

CUSOLVER_STATUS_IRS_INFOS_NOT_INITIALIZED | The information structure gels_irs_infos was not created.

CUSOLVER_STATUS_ALLOC_FAILED | CPU memory allocation failed, most likely during the allocation of the residual array that store the residual norms.

### 2.4.2.16. cusolverDn<t>ormqr()

These helper functions calculate the size of work buffers needed.

cusolverStatus_t
cusolverDnSormqr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasOperation_t trans,
    int m,
    int n,
    int k,
    const float *A,
    int lda,
    const float *tau,
    const float *C,
    int ldc,
    int *lwork);

cusolverStatus_t
cusolverDnDormqr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasOperation_t trans,
    int m,
    int n,
    int k,
    const double *A,
    int lda,
    const double *tau,
    const double *C,
    int ldc,
    int *lwork);

cusolverStatus_t
cusolverDnCunmqr_bufferSize(
    cusolverDnHandle_t handle,
The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t

cusolverDnSormqr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasOperation_t trans,
    int m,
    int n,
    int k,
    const float *A,
    int lda,
    const float *tau,
    float *C,
    int ldc,
    float *work,
    int *devInfo);

cusolverStatus_t

cusolverDnDormqr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasOperation_t trans,
    int m,
    int n,
    int k,
    const double *A,
    int lda,
    const double *tau,
    double *C,
    int ldc,
    double *work,
The C and Z data types are complex valued single and double precision, respectively.

```c
int lwork,
int *devInfo);
```

This function overwrites $m \times n$ matrix $C$ by

$$C = \begin{cases} \text{op}(Q) \cdot C & \text{if } \text{side} == \text{CUBLAS_SIDE_LEFT} \\ C \cdot \text{op}(Q) & \text{if } \text{side} == \text{CUBLAS_SIDE_RIGHT} \end{cases}$$

The operation of $Q$ is defined by

$$\text{op}(Q) = \begin{cases} Q & \text{if } \text{transa} == \text{CUBLAS_OP_N} \\ Q^T & \text{if } \text{transa} == \text{CUBLAS_OP_T} \\ Q^H & \text{if } \text{transa} == \text{CUBLAS_OP_C} \end{cases}$$

$Q$ is a unitary matrix formed by a sequence of elementary reflection vectors from QR factorization ($\text{geqrt}$) of $A$.

$Q = \text{H}(1) \cdot \text{H}(2) \ldots \text{H}(k)$

$Q$ is of order $m$ if $\text{side} = \text{CUBLAS_SIDE_LEFT}$ and of order $n$ if $\text{side} = \text{CUBLAS_SIDE_RIGHT}$. 
The user has to provide working space which is pointed by input parameter `work`. The input parameter `lwork` is size of the working space, and it is returned by `geqrf_bufferSize()` or `ormqr_bufferSize()`.

If output parameter `devInfo = -i` (less than zero), the `i`-th parameter is wrong (not counting handle).

The user can combine `geqrf`, `ormqr` and `trsm` to complete a linear solver or a least-square solver. Please refer to appendix C.1.

**API of ormqr**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>Handle to the cuSolverDn library context.</td>
</tr>
<tr>
<td>side</td>
<td>host</td>
<td>input</td>
<td>Indicates if matrix $Q$ is on the left or right of $C$.</td>
</tr>
<tr>
<td>trans</td>
<td>host</td>
<td>input</td>
<td>Operation $\text{op}(Q)$ that is non- or (conj.) transpose.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>Number of columns of matrix $C$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>Number of rows of matrix $C$.</td>
</tr>
<tr>
<td>k</td>
<td>host</td>
<td>input</td>
<td>Number of elementary reflections whose product defines the matrix $Q$.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>$&lt;$type$&gt;$ array of dimension $\text{lda} \times k$ with $\text{lda}$ is not less than $\max(1,m)$. The matrix $A$ is from $\text{geqrf}$, so $i$-th column contains elementary reflection vector.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of two-dimensional array used to store matrix $A$. if <code>side</code> is $\text{CUBLAS_SIDE_LEFT}$, $\text{lda} \geq \max(1,m)$; if <code>side</code> is $\text{CUBLAS_SIDE_RIGHT}$, $\text{lda} \geq \max(1,n)$.</td>
</tr>
<tr>
<td>tau</td>
<td>device</td>
<td>output</td>
<td>$&lt;$type$&gt;$ array of dimension at least $\min(m,n)$. The vector $\tau$ is from $\text{geqrf}$, so $\tau(i)$ is the scalar of $i$-th elementary reflection vector.</td>
</tr>
<tr>
<td>C</td>
<td>device</td>
<td>in/out</td>
<td>$&lt;$type$&gt;$ array of size $\text{ldc} \times n$. On exit, $C$ is overwritten by $\text{op}(Q) \times C$.</td>
</tr>
<tr>
<td>ldc</td>
<td>host</td>
<td>input</td>
<td>Leading dimension of two-dimensional array of matrix $C$. $\text{ldc} \geq \max(1,m)$.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>Working space, $&lt;$type$&gt;$ array of size <code>lwork</code>.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>Size of working array <code>work</code>.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>If <code>devInfo = 0</code>, the <code>ormqr</code> is successful. If <code>devInfo = -i</code>, the <code>i</code>-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

<p>| CUSOLVER_STATUS_SUCCESS | The operation completed successfully. |</p>
<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_NOT_INITIALIZED</th>
<th>The library was not initialized.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>Invalid parameters were passed ((m, n\leq 0) or wrong (lda) or (ldc)).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>The device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>An internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.2.17. `cusolverDn<t>orgqr()`

These helper functions calculate the size of work buffers needed.

```c
 cusolverStatus_t
cusolverDnSorgqr_bufferSize(  
    cusolverDnHandle_t handle,  
    int m,  
    int n,  
    int k,  
    const float *A,  
    int lda,  
    const float *tau,  
    int *lwork);

cusolverStatus_t
cusolverDnDorgqr_bufferSize(  
    cusolverDnHandle_t handle,  
    int m,  
    int n,  
    int k,  
    const double *A,  
    int lda,  
    const double *tau,  
    int *lwork);

cusolverStatus_t
cusolverDnCungqr_bufferSize(  
    cusolverDnHandle_t handle,  
    int m,  
    int n,  
    int k,  
    const cuComplex *A,  
    int lda,  
    const cuComplex *tau,  
    int *lwork);

cusolverStatus_t
cusolverDnZungqr_bufferSize(  
    cusolverDnHandle_t handle,  
    int m,  
    int n,  
    int k,  
    const cuDoubleComplex *A,  
    int lda,  
    const cuDoubleComplex *tau,  
    int *lwork);
```

The S and D data types are real valued single and double precision, respectively.
The C and Z data types are complex valued single and double precision, respectively.

This function overwrites \( m \times n \) matrix \( A \) by

\[
Q = H(1)H(2)\ldots H(k)
\]

where \( Q \) is a unitary matrix formed by a sequence of elementary reflection vectors stored in \( A \).
The user has to provide working space which is pointed by input parameter `work`. The input parameter `lwork` is size of the working space, and it is returned by `orgqr_bufferSize()`.

If output parameter `devInfo = -i` (less than zero), the `i`-th parameter is wrong (not counting handle).

The user can combine `geqrf`, `orgqr` to complete orthogonalization. Please refer to appendix C.2.

**API of `ormqr`**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $Q$. $m \geq 0$;</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $Q$. $m \geq n \geq 0$;</td>
</tr>
<tr>
<td><code>k</code></td>
<td>host</td>
<td>input</td>
<td>number of elementary reflections whose product defines the matrix $Q$. $n \geq k \geq 0$;</td>
</tr>
<tr>
<td><code>A</code></td>
<td>device</td>
<td>in/out</td>
<td>$\langle$type$&gt;$ array of dimension $lda \times n$ with $lda$ is not less than $\max(1,m)$. $i$-th column of $A$ contains elementary reflection vector.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$. $lda \geq \max(1,m)$.</td>
</tr>
<tr>
<td><code>tau</code></td>
<td>device</td>
<td>output</td>
<td>$\langle$type$&gt;$ array of dimension $k$. $\tau(i)$ is the scalar of $i$-th elementary reflection vector.</td>
</tr>
<tr>
<td><code>work</code></td>
<td>device</td>
<td>in/out</td>
<td>working space, $\langle$type$&gt;$ array of size $lwork$.</td>
</tr>
<tr>
<td><code>lwork</code></td>
<td>host</td>
<td>input</td>
<td>size of working array $work$.</td>
</tr>
<tr>
<td><code>devInfo</code></td>
<td>device</td>
<td>output</td>
<td>if $info = 0$, the <code>orgqr</code> is successful. if $info = -i$, the <code>i</code>-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ($m,n,k&lt;0,n&gt;m$, $k&gt;n$ or $lda&lt;m$).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.2.18. cusolverDn<t>sytrf()

These helper functions calculate the size of the needed buffers.
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusolverDnSsytrf_bufferSize</td>
<td>Calculates the buffer size for the symmetric triangular factorization</td>
<td>S, D</td>
</tr>
<tr>
<td>cusolverDnDsytrf_bufferSize</td>
<td>Calculates the buffer size for the symmetric triangular factorization</td>
<td>S, D</td>
</tr>
<tr>
<td>cusolverDnCsytrf_bufferSize</td>
<td>Calculates the buffer size for the complex symmetric triangular factorization</td>
<td>C, Z</td>
</tr>
<tr>
<td>cusolverDnZsytrf_bufferSize</td>
<td>Calculates the buffer size for the complex symmetric triangular factorization</td>
<td>C, Z</td>
</tr>
</tbody>
</table>

The S and D data types are real valued single and double precision, respectively.

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusolverDnSsytrf</td>
<td>Performs the symmetric triangular factorization</td>
<td>S</td>
</tr>
<tr>
<td>cusolverDnDsytrf</td>
<td>Performs the symmetric triangular factorization</td>
<td>D</td>
</tr>
<tr>
<td>cusolverDnCsytrf</td>
<td>Performs the complex symmetric triangular factorization</td>
<td>C</td>
</tr>
<tr>
<td>cusolverDnZsytrf</td>
<td>Performs the complex symmetric triangular factorization</td>
<td>Z</td>
</tr>
</tbody>
</table>

The C and Z data types are complex valued single and double precision, respectively.
This function computes the Bunch-Kaufman factorization of an \( n \times n \) symmetric indefinite matrix \( A \).

If input parameter \( \text{uplo} \) is \text{CUBLAS\_FILL\_MODE\_LOWER}, only lower triangular part of \( A \) is processed, and replaced by lower triangular factor \( L \) and block diagonal matrix \( D \). Each block of \( D \) is either 1x1 or 2x2 block, depending on pivoting.

\[
P \cdot A \cdot P^T = L \cdot D \cdot L^T
\]

If input parameter \( \text{uplo} \) is \text{CUBLAS\_FILL\_MODE\_UPPER}, only upper triangular part of \( A \) is processed, and replaced by upper triangular factor \( U \) and block diagonal matrix \( D \).

\[
P \cdot A \cdot P^T = U \cdot D \cdot U^T
\]

The user has to provide working space which is pointed by input parameter \( \text{work} \). The input parameter \( \text{lwork} \) is size of the working space, and it is returned by \text{sytrf\_bufferSize()}.

If Bunch-Kaufman factorization failed, i.e. \( A \) is singular. The output parameter \( \text{devInfo} = i \) would indicate \( D(i,i) = 0 \).

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

The output parameter \( \text{devIpiv} \) contains pivoting sequence. If \( \text{devIpiv}(i) = k > 0 \), \( D(i,i) \) is 1x1 block, and \( i \)-th row/column of \( A \) is interchanged with \( k \)-th row/column of \( A \). If \text{uplo} is \text{CUBLAS\_FILL\_MODE\_UPPER} and \( \text{devIpiv}(i-1) = \text{devIpiv}(i) = -m < 0 \), \( D(i-1:i-1,i-1:i) \) is a 2x2 block, and \( (i-1) \)-th row/column is interchanged with \( m \)-th row/column. If \text{uplo} is \text{CUBLAS\_FILL\_MODE\_LOWER} and \( \text{devIpiv}(i+1) = \text{devIpiv}(i) = -m < 0 \), \( D(i:i+1,i:i+1) \) is a 2x2 block, and \( (i+1) \)-th row/column is interchanged with \( m \)-th row/column.

\textbf{API of sytrf}

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix ( A ) lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( A )</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension ( \text{lda} \times n ) with ( \text{lda} ) is not less than ( \max(1,n) ).</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda</td>
<td>host</td>
<td>input leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>ipiv</td>
<td>device</td>
<td>output array of size at least n, containing pivot indices.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input size of working space work.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output if devInfo = 0, the LU factorization is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo = i, the D(i,i) = 0.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed [n<0 or lda<max(1,n)].
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.

### 2.4.2.19. cusolverDn<t>potrfBatched()

The S and D data types are real valued single and double precision, respectively.

```c
 cusolverDnSpotrfBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    float *Aarray[],
    int lda,
    int *infoArray,
    int batchSize);
```

```c
 cusolverDnDpotrfBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    double *Aarray[],
    int lda,
    int *infoArray,
    int batchSize);
```

The C and Z data types are complex valued single and double precision, respectively.

```c
 cusolverDnCpotrfBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuComplex *Aarray[],
    int lda,
    int *infoArray,
    int batchSize);
```

```c
 cusolverDnZpotrfBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *Aarray[],
    int lda,
    int *infoArray,
    int batchSize);
```
This function computes the Cholesky factorization of a sequence of Hermitian positive-definite matrices.

Each AArray[i] for i=0,1,..., batchSize-1 is a n×n Hermitian matrix, only lower or upper part is meaningful. The input parameter uplo indicates which part of the matrix is used.

If input parameter uplo is CUBLAS_FILL_MODE_LOWER, only lower triangular part of A is processed, and replaced by lower triangular Cholesky factor L.

\[ A = L \cdot L^H \]

If input parameter uplo is CUBLAS_FILL_MODE_UPPER, only upper triangular part of A is processed, and replaced by upper triangular Cholesky factor U.

\[ A = U^H \cdot U \]

If Cholesky factorization failed, i.e. some leading minor of A is not positive definite, or equivalently some diagonal elements of L or U is not a real number. The output parameter infoArray would indicate smallest leading minor of A which is not positive definite.

infoArray is an integer array of size batchSize. If potrfBatched returns CUSOLVER_STATUS_INVALID_VALUE, infoArray[0] = -i (less than zero), meaning that the i-th parameter is wrong [not counting handle]. If potrfBatched returns CUSOLVER_STATUS_SUCCESS but infoArray[i] = k is positive, then i-th matrix is not positive definite and the Cholesky factorization failed at row k.

Remark: the other part of A is used as a workspace. For example, if uplo is CUBLAS_FILL_MODE_UPPER, upper triangle of A contains choleksy factor U and lower triangle of A is destroyed after potrfBatched.

**API of potrfBatched**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if lower or upper part is stored, the other part is used as a workspace.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix A.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Aarray</th>
<th>device</th>
<th>in/out</th>
<th>array of pointers to &lt;type&gt; array of dimension lda * n with lda is not less than max(1,n).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>lda</th>
<th>host</th>
<th>input</th>
<th>leading dimension of two-dimensional array used to store each matrix Aarray[i].</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>infoArray</th>
<th>device</th>
<th>output</th>
<th>array of size batchSize. infoArray[i] contains information of factorization of Aarray[i]. If potrfBatched returns CUSOLVER_STATUS_INVALID_VALUE, infoArray[0] = -i (less than zero) means the i-th parameter is wrong (not counting handle). If potrfBatched returns CUSOLVER_STATUS_SUCCESS, infoArray[i] = 0 means the Cholesky factorization of i-th matrix is successful, and infoArray[i] = k means the leading submatrix of order k of i-th matrix is not positive definite.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>batchSize</th>
<th>host</th>
<th>input</th>
<th>number of pointers in Aarray.</th>
</tr>
</thead>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**: invalid parameters were passed (n<0 or lda<max(1,n) or batchSize<1).
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.

### 2.4.2.20. cusolverDn<t>potrsBatched()

```c
cusolverStatus_t cusolverDnSpotrsBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    int nrhs,
    float *Aarray[],
    int lda,
    float *Barray[],
    int ldb,
    int *info,
    int batchSize);
```

```c
cusolverStatus_t cusolverDnDpotrsBatched(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    int nrhs,
    double *Aarray[],
    int lda,
    double *Barray[],
```
This function solves a sequence of linear systems

\[ A[i] X[i] = B[i] \]

where each \( A[i] \) for \( i=0,1,\ldots, \) batchSize-1 is an \( n \times n \) Hermitian matrix, only lower or upper part is meaningful. The input parameter \( \text{uplo} \) indicates which part of the matrix is used.

The user has to call \text{potrfBatched} first to factorize matrix \( A[i] \). If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_LOWER}, \( A \) is lower triangular Cholesky factor \( L \) corresponding to \( A = L^* L^H \). If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_UPPER}, \( A \) is upper triangular Cholesky factor \( U \) corresponding to \( A = U^H U \).

The operation is in-place, i.e. matrix \( X \) overwrites matrix \( B \) with the same leading dimension \( \text{ldb} \).

The output parameter \( \text{info} \) is a scalar. If \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

Remark 1: only \( \text{nrhs}=1 \) is supported.

Remark 2: \( \text{infoArray} \) from \text{potrfBatched} indicates if the matrix is positive definite. \( \text{info} \) from \text{potrsBatched} only shows which input parameter is wrong (not counting handle).

Remark 3: the other part of \( A \) is used as a workspace. For example, if \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_UPPER}, upper triangle of \( A \) contains cholesky factor \( U \) and lower triangle of \( A \) is destroyed after \text{potrsBatched}.

API of \text{potrsBatched}
<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolveDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix A lower or upper part is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix X and B.</td>
</tr>
<tr>
<td>Aarray</td>
<td>device</td>
<td>in/out</td>
<td>array of pointers to &lt;type&gt; array of dimension lda * n with lda is not less than max(1,n). Aarray[i] is either lower cholesky factor L or upper Cholesky factor U.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store each matrix Aarray[i].</td>
</tr>
<tr>
<td>Barray</td>
<td>device</td>
<td>in/out</td>
<td>array of pointers to &lt;type&gt; array of dimension ldb * nrhs. ldb is not less than max(1,n). As an input, Barray[i] is right hand side matrix. As an output, Barray[i] is the solution matrix.</td>
</tr>
<tr>
<td>ldb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store each matrix Barray[i].</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, all parameters are correct. if info = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
<tr>
<td>batchSize</td>
<td>host</td>
<td>input</td>
<td>number of pointers in Aarray.</td>
</tr>
</tbody>
</table>

### Status Returned

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**: invalid parameters were passed \([n<0, \text{nrhs}<0, lda<\text{max}(1,n), ldb<\text{max}(1,n) \text{ or batchSize}<0]\). |
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.

### 2.4.3. Dense Eigenvalue Solver Reference (legacy)

This chapter describes eigenvalue solver API of cuSolverDN, including bidiagonalization and SVD.

#### 2.4.3.1. cusolverDn<t>gebrd()

These helper functions calculate the size of work buffers needed.

```c
 cusolverStatus_t
cusolverDnSgebrd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
```
int n,
int *Lwork);
cusolverStatus_t
cusolverDnDgebrd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int *Lwork);

cusolverStatus_t
cusolverDnCgebrd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int *Lwork);

cusolverStatus_t
cusolverDnZgebrd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int *Lwork);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSgebrd(cusolverDnHandle_t handle,
    int m,
    int n,
    float *A,
    int lda,
    float *D,
    float *E,
    float *TAUQ,
    float *TAUP,
    float *Work,
    int Lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnDgebrd(cusolverDnHandle_t handle,
    int m,
    int n,
    double *A,
    int lda,
    double *D,
    double *E,
    double *TAUQ,
    double *TAUP,
    double *Work,
    int Lwork,
    int *devInfo);

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnCgebrd(cusolverDnHandle_t handle,
    int m,
    int n,
    cuComplex *A,
This function reduces a general $m \times n$ matrix $A$ to a real upper or lower bidiagonal form $B$ by an orthogonal transformation: $Q^H A P = B$

If $m \geq n$, $B$ is upper bidiagonal; if $m < n$, $B$ is lower bidiagonal.

The matrix $Q$ and $P$ are overwritten into matrix $A$ in the following sense:

if $m \geq n$, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix $B$; the elements below the diagonal, with the array $\text{TAUQ}$, represent the orthogonal matrix $Q$ as a product of elementary reflectors, and the elements above the first superdiagonal, with the array $\text{TAUP}$, represent the orthogonal matrix $P$ as a product of elementary reflectors.

if $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix $B$; the elements below the first subdiagonal, with the array $\text{TAUQ}$, represent the orthogonal matrix $Q$ as a product of elementary reflectors, and the elements above the diagonal, with the array $\text{TAUP}$, represent the orthogonal matrix $P$ as a product of elementary reflectors.

The user has to provide working space which is pointed by input parameter $\text{Work}$. The input parameter $Lwork$ is size of the working space, and it is returned by $\text{gebrd_bufferSize()}$.

If output parameter $\text{devInfo} = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle).

Remark: $\text{gebrd}$ only supports $m \geq n$.

**API of $\text{gebrd}$**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>$m$</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>$n$</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>in/out</td>
<td>$&lt;$type$&gt;$ array of dimension $\text{lda} \times n$ with $\text{lda}$ is not less than max(1, $n$).</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>D</td>
<td>device</td>
<td>output</td>
<td>real array of dimension min(m,n). The diagonal elements of the bidiagonal matrix B: D(i) = A(i,i).</td>
</tr>
<tr>
<td>E</td>
<td>device</td>
<td>output</td>
<td>real array of dimension min(m,n). The off-diagonal elements of the bidiagonal matrix B: if m&gt;=n, E(i) = A(i,i+1) for i = 1,2,...,n-1; if m&lt;n, E(i) = A(i+1,i) for i = 1,2,...,m-1.</td>
</tr>
<tr>
<td>TAUQ</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension min(m,n). The scalar factors of the elementary reflectors which represent the orthogonal matrix Q.</td>
</tr>
<tr>
<td>TAUP</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension min(m,n). The scalar factors of the elementary reflectors which represent the orthogonal matrix P.</td>
</tr>
<tr>
<td>Work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size Lwork.</td>
</tr>
<tr>
<td>Lwork</td>
<td>host</td>
<td>input</td>
<td>size of Work, returned by gebrd_bufferSize.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the operation is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed [m, n<0, or lda<max(1,m)].
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.

### 2.4.3.2. cusolverDn<t>orgbr() 

These helper functions calculate the size of work buffers needed.

```c

cusolverDnSorgbr_bufferSize(
    cusolverDnHandle_t handle, 
    cublasSideMode_t side, 
    int m, 
    int n, 
    int k, 
    const float *A, 
    int lda,
```

cuSOLVER Library
const float *tau,
int *lwork);
cusolverStatus_t
cusolverDnDorgbr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    int m,
    int n,
    int k,
    const double *A,
    int lda,
    const double *tau,
    int *lwork);

cusolverStatus_t
cusolverDnCungbr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    int m,
    int n,
    int k,
    const cuComplex *A,
    int lda,
    const cuComplex *tau,
    int *lwork);

cusolverStatus_t
cusolverDnZungbr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    int m,
    int n,
    int k,
    const cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *tau,
    int *lwork);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSorgbr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    int m,
    int n,
    int k,
    float *A,
    int lda,
    const float *tau,
    float *work,
    int lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnDorgbr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    int m,
The C and Z data types are complex valued single and double precision, respectively.

This function generates one of the unitary matrices $Q$ or $P^{*H}$ determined by \texttt{gebrd} when reducing a matrix $A$ to bidiagonal form: $Q^{H} * A * P = B$

$Q$ and $P^{*H}$ are defined as products of elementary reflectors $H(i)$ or $G(i)$ respectively.

The user has to provide working space which is pointed by input parameter \texttt{work}. The input parameter \texttt{lwork} is size of the working space, and it is returned by \texttt{orgbr\_bufferSize()}.

If output parameter \texttt{devInfo} = -i (less than zero), the i-th parameter is wrong [not counting handle].

**API of orgbr**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>side</td>
<td>host</td>
<td>input</td>
<td>if side = CUBLAS_SIDE_LEFT, generate $Q$, if side = CUBLAS_SIDE_RIGHT, generate $P^{*T}$.</td>
</tr>
</tbody>
</table>
### Using the CUSOLVER API

#### cuSOLVER Library

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix Q or $P^T$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>if ( \text{side} = \text{CUBLAS_SIDE_LEFT}, m \geq n \geq \min(m,k) ). if ( \text{side} = \text{CUBLAS_SIDE_RIGHT}, n \geq m \geq \min(n,k) ).</td>
</tr>
<tr>
<td>k</td>
<td>host</td>
<td>input</td>
<td>if ( \text{side} = \text{CUBLAS_SIDE_LEFT} ), the number of columns in the original ( m ) by ( k ) matrix reduced by \text{gebrd}. if ( \text{side} = \text{CUBLAS_SIDE_RIGHT} ), the number of rows in the original ( k ) by ( n ) matrix reduced by \text{gebrd}.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension ( \text{lda} \times n ). On entry, the vectors which define the elementary reflectors, as returned by \text{gebrd}. On exit, the ( m )-by-( n ) matrix ( Q ) or ( P^T ).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ). lda \geq \max(1,m);</td>
</tr>
<tr>
<td>tau</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ( \min(m,k) ) if ( \text{side} = \text{CUBLAS_SIDE_LEFT} ); of dimension ( \min(n,k) ) if ( \text{side} = \text{CUBLAS_SIDE_RIGHT} ); ( \tau[i] ) must contain the scalar factor of the elementary reflector ( H[i] ) or ( G[i] ), which determines ( Q ) or ( P^T ), as returned by \text{gebrd} in its array argument \text{TAUQ} ) or \text{TAUP}.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size ( \text{lwork} ).</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of working array ( \text{work} ).</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if ( \text{info} = 0 ), the \text{ormqr} is successful. if ( \text{info} = -i ), the ( i )-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

#### Status Returned

- **CUSOLVER\_STATUS\_SUCCESS**
  - the operation completed successfully.
- **CUSOLVER\_STATUS\_NOT\_INITIALIZED**
  - the library was not initialized.
- **CUSOLVER\_STATUS\_INVALID\_VALUE**
  - invalid parameters were passed \( [m, n<0 \text{ or wrong } \text{lda}] \).
- **CUSOLVER\_STATUS\_ARCH\_MISMATCH**
  - the device only supports compute capability 2.0 and above.
- **CUSOLVER\_STATUS\_INTERNAL\_ERROR**
  - an internal operation failed.

#### 2.4.3.3. cuSolverDn<t>sytrd() 

These helper functions calculate the size of work buffers needed.

```c
 cusolverStatus_t 
cusolverDnSsytrd_bufferSize(``
```
The S and D data types are real valued single and double precision, respectively.
int *devInfo);

cusolverStatus_t
cusolverDnDsytrd(
    cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
double *A,
    int lda,
double *d,
double *e,
double *tau,
double *work,
    int lwork,
    int *devInfo);

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnChetrd(
    cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
cuComplex *A,
    int lda,
    float *d,
    float *e,
cuComplex *tau,
cuComplex *work,
    int lwork,
    int *devInfo);

cusolverStatus_t CUDENSEAPI cusolverDnZhetrd(
    cusolverDnHandle_t handle,
cublasFillMode_t uplo,
    int n,
cuDoubleComplex *A,
    int lda,
double *d,
double *e,
cuDoubleComplex *tau,
cuDoubleComplex *work,
    int lwork,
    int *devInfo);

This function reduces a general symmetric (Hermitian) \( n \times n \) matrix \( A \) to real symmetric tridiagonal form \( T \) by an orthogonal transformation: \( Q^H A Q = T \)

As an output, \( A \) contains \( T \) and householder reflection vectors. If \( \text{uplo} = \text{CUBLAS_FILL_MODE_UPPER} \), the diagonal and first superdiagonal of \( A \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements above the first superdiagonal, with the array \( \text{tau} \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors; if \( \text{uplo} = \text{CUBLAS_FILL_MODE_LOWER} \), the diagonal and first subdiagonal of \( A \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements below the first subdiagonal, with the array \( \text{tau} \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors.

The user has to provide working space which is pointed by input parameter \( \text{work} \). The input parameter \( \text{lwork} \) is size of the working space, and it is returned by \( \text{sytrd_bufferSize()} \).
If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle).

**API of sytrd**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of A is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows (columns) of matrix A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) array of dimension (\text{lda} \times n) with (\text{lda}) is not less than (\max(1,n)). If uplo = CUBLAS_FILL_MODE_UPPER, the leading (n)-by-(n) upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If uplo = CUBLAS_FILL_MODE_LOWER, the leading (n)-by-(n) lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. On exit, A is overwritten by T and householder reflection vectors.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. (\text{lda} \geq \max(1,n)).</td>
</tr>
<tr>
<td>D</td>
<td>device</td>
<td>output</td>
<td>real array of dimension n. The diagonal elements of the tridiagonal matrix T: (D(i) = A(i,i)).</td>
</tr>
<tr>
<td>E</td>
<td>device</td>
<td>output</td>
<td>real array of dimension ((n-1)). The off-diagonal elements of the tridiagonal matrix T: if uplo = CUBLAS_FILL_MODE_UPPER, (E(i) = A(i,i+1)). if uplo = CUBLAS_FILL_MODE_LOWER (E(i) = A(i+1,i)).</td>
</tr>
<tr>
<td>tau</td>
<td>device</td>
<td>output</td>
<td>(&lt;\text{type}&gt;) array of dimension ((n-1)). The scalar factors of the elementary reflectors which represent the orthogonal matrix Q.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, (&lt;\text{type}&gt;) array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of work, returned by sytrd_bufferSize.</td>
</tr>
</tbody>
</table>
| devInfo   | device | output | if devInfo = 0, the operation is successful. if devInfo = -i, the i-
Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [n&lt;0, or lda&lt;max(1,n), or uplo IS not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.3.4. **cusolverDn<t>ormtr()**

These helper functions calculate the size of work buffers needed.

```c
 cusolverStatus_t
cusolverDnSormtr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
    const float *A,
    int lda,
    const float *tau,
    const float *C,
    int ldc,
    int *lwork);

cusolverStatus_t
cusolverDnDormtr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
    const double *A,
    int lda,
    const double *tau,
    const double *C,
    int ldc,
    int *lwork);

cusolverStatus_t
cusolverDnCunmtr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
```
Using the CUSOLVER API

```c
const cuComplex *A,
int lda,
const cuComplex *tau,
const cuComplex *C,
int ldc,
int *lwork);

cusolverStatus_t
cusolverDnZunmtr_bufferSize(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
    const cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *tau,
    const cuDoubleComplex *C,
    int ldc,
    int *lwork);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSormtr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
    float *A,
    int lda,
    float *tau,
    float *C,
    int ldc,
    float *work,
    int lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnDormtr(
    cusolverDnHandle_t handle,
    cublasSideMode_t side,
    cublasFillMode_t uplo,
    cublasOperation_t trans,
    int m,
    int n,
    double *A,
    int lda,
    double *tau,
    double *C,
    int ldc,
    double *work,
    int lwork,
    int *devInfo);
```
The C and Z data types are complex valued single and double precision, respectively.

The user has to provide working space which is pointed by input parameter work. The input parameter lwork is size of the working space, and it is returned by ormtr_bufferSize().

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle).

**API of ormtr**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
</table>

In the CUSOLVER API, the functions `cusolverDnCunmtr` and `cusolverDnZunmtr` are used to apply a unitary matrix formed by a sequence of elementary reflection vectors to a matrix. These functions are part of the cuSOLVER library, which is a part of the cuSOLVER Library.

The functions `cusolverDnCunmtr` and `cusolverDnZunmtr` take the following parameters:

- `handle`: A handle that represents a cuSOLVER context.
- `side`: Indicates the side of the matrix multiplication.
- `uplo`: Indicates whether the upper or lower triangle of the matrix is to be used.
- `trans`: Specifies the operation on the matrix.
- `m`: Number of rows of the matrix.
- `n`: Number of columns of the matrix.
- `A`: Pointer to the matrix to be multiplied.
- `lda`: Leading dimension of the matrix.
- `tau`: Pointer to the array containing the elementary reflection vectors.
- `C`: Pointer to the matrix to be overwritten.
- `ldc`: Leading dimension of the overwritten matrix.
- `work`: Pointer to the working space.
- `lwork`: Size of the working space.
- `devInfo`: A pointer to an integer that contains information about the success or failure of the function.

The function overwrites an \( m \times n \) matrix \( C \) by

\[
C = \begin{cases} 
\text{op}(Q) \cdot C & \text{if } \text{side} == \text{CUBLAS_SIDE_LEFT} \\
C \cdot \text{op}(Q) & \text{if } \text{side} == \text{CUBLAS_SIDE_RIGHT}
\end{cases}
\]

where \( Q \) is a unitary matrix formed by a sequence of elementary reflection vectors from `sytrd`. The operation on \( Q \) is defined by

\[
\text{op}(Q) = \begin{cases} 
Q & \text{if } \text{transa} == \text{CUBLAS_OP_N} \\
Q^T & \text{if } \text{transa} == \text{CUBLAS_OP_T} \\
Q^H & \text{if } \text{transa} == \text{CUBLAS_OP_C}
\end{cases}
\]
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>side</td>
<td>host</td>
<td>input</td>
<td>side = CUBLAS_SIDE_LEFT, apply Q or Q<strong>T from the Left; side = CUBLAS_SIDE_RIGHT, apply Q or Q</strong>T from the Right.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A contains elementary reflectors from sytrd. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A contains elementary reflectors from sytrd.</td>
</tr>
<tr>
<td>trans</td>
<td>host</td>
<td>input</td>
<td>operation op(Q) that is non- or (conj.) transpose.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix C.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix C.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * m if side = CUBLAS_SIDE_LEFT; lda * n if side = CUBLAS_SIDE_RIGHT. The matrix A from sytrd contains the elementary reflectors.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. if side is CUBLAS_SIDE_LEFT, lda &gt;= max(1,m); if side is CUBLAS_SIDE_RIGHT, lda &gt;= max(1,n).</td>
</tr>
<tr>
<td>tau</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension (m-1) if side is CUBLAS_SIDE_LEFT; of dimension (n-1) if side is CUBLAS_SIDE_RIGHT. The vector tau is from sytrd, so tau(i) is the scalar of i-th elementary reflection vector.</td>
</tr>
<tr>
<td>C</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of size ldc * n. On exit, C is overwritten by op(Q) * C or C*op(Q).</td>
</tr>
<tr>
<td>ldc</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array of matrix C. ldc &gt;= max(1,m).</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of working array work.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the ormqr is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_ARCH_MISMATCH | the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR | an internal operation failed.

### 2.4.3.5. `cusolverDn<t>orgtr()`

These helper functions calculate the size of work buffers needed.

```c
cusolverStatus_t
cusolverDnSorgtr_bufferSize(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n, 
    const float *A, 
    int lda,
    const float *tau,
    int *lwork);

cusolverStatus_t
cusolverDnDorgtr_bufferSize(
    cusolverDnHandle_t handle, 
    cublasFillMode_t uplo, 
    int n,
    const double *A, 
    int lda,
    const double *tau,
    int *lwork);

cusolverStatus_t
cusolverDnCungtr_bufferSize(
    cusolverDnHandle_t handle, 
    cublasFillMode_t uplo, 
    int n,
    const cuComplex *A, 
    int lda,
    const cuComplex *tau,
    int *lwork);

cusolverStatus_t
cusolverDnZungtr_bufferSize(
    cusolverDnHandle_t handle, 
    cublasFillMode_t uplo, 
    int n,
    const cuDoubleComplex *A, 
    int lda,
    const cuDoubleComplex *tau,
    int *lwork);
```

The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnSorgtr(
    cusolverDnHandle_t handle, 
    cublasFillMode_t uplo, 
    int n,
    float *A, 
    int lda,
    const float *tau,
```
The C and Z data types are complex valued single and double precision, respectively.

This function generates a unitary matrix $Q$ which is defined as the product of $n$-1 elementary reflectors of order $n$, as returned by sytrd:

The user has to provide working space which is pointed by input parameter work. The input parameter lwork is size of the working space, and it is returned by orgtr_bufferSize().

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong [not counting handle].

### API of orgtr

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of $A$ contains elementary reflectors from sytrd. uplo =</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

CUSOLVER_FILL_MODE_UPPER: Upper triangle of A contains elementary reflectors from sytrd.

<table>
<thead>
<tr>
<th>n</th>
<th>host</th>
<th>input</th>
<th>number of rows (columns) of matrix Q.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n On entry, matrix A from sytrd contains the elementary reflectors. On exit, matrix A contains the n-by-n orthogonal matrix Q.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. lda &gt;= max(1,n).</td>
</tr>
<tr>
<td>tau</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension (n-1) tau(i) is the scalar of i-th elementary reflection vector.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of working array work.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the orgtr is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [n&lt;0 or wrong lda ].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.3.6. cusolverDn<t>gesvd()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

cusolverStatus_t
cusolverDnSgesvd_bufferSize(
cusolverDnHandle_t handle,
int m,
int n,
int *lwork );
cusolverStatus_t
cusolverDnDgesvd_bufferSize(
cusolverDnHandle_t handle,
int m,
int n,
int *lwork );
cusolverStatus_t
cusolverDnCgesvd_bufferSize(
cusolverDnHandle_t handle,
int m,
int n,
The S and D data types are real valued single and double precision, respectively.

```c
int *lwork );
cusolverStatus_t
cusolverDnZgesvd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int *lwork );
```

The C and Z data types are complex valued single and double precision, respectively.

```c
int *lwork );
cusolverStatus_t
cusolverDnCgesvd (  
cusolverDnHandle_t handle,  
signed char jobu,  
signed char jobvt,  
int m,  
int n,  
cuComplex *A,  
int lda,  
cuComplex *S,  
cuComplex *U,  
int ldu,  
cuComplex *VT,  
int ldvt,  
cuComplex *work,  
int lwork,  
cuComplex *rwork,  
int *devInfo);
```

```c
int *lwork );
cusolverStatus_t
cusolverDnDgesvd (  
cusolverDnHandle_t handle,  
signed char jobu,  
signed char jobvt,  
int m,  
int n,  
double *A,  
int lda,  
double *S,  
double *U,  
int ldu,  
double *VT,  
int ldvt,  
double *work,  
int lwork,  
double *rwork,  
int *devInfo);
```
using the CUSOLVER API

cuDoubleComplex *work,
int lwork,
float *rwork,
int *devInfo);

cusolverStatus_t

cusolverDnZgesvd (|
cusolverDnHandle_t handle,
signed char jobu,
signed char jobvt,
int m,
int n,
cuDoubeComplex *A,
int lda,
double *S,
cuDoubeComplex *U,
int ldu,
cuDoubeComplex *VT,
int ldvt,
cuDoubeComplex *work,
int lwork,
double *rwork,
int *devInfo);

This function computes the singular value decomposition (SVD) of a \( m \times n \) matrix \( A \) and corresponding the left and/or right singular vectors. The SVD is written

\[
A = U \Sigma V^H
\]

where \( \Sigma \) is an \( m \times n \) matrix which is zero except for its \( \min(m, n) \) diagonal elements, \( U \) is an \( m \times m \) unitary matrix, and \( V \) is an \( n \times n \) unitary matrix. The diagonal elements of \( \Sigma \) are the singular values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m, n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

The user has to provide working space which is pointed by input parameter \( \text{work} \). The input parameter \( \text{lwork} \) is size of the working space, and it is returned by \( \text{gesvd\_bufferSize}() \).

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle). If \( \text{bdsqr} \) did not converge, \( \text{devInfo} \) specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.

The \( \text{rwork} \) is real array of dimension \( \min(m,n)-1 \). If \( \text{devInfo} > 0 \) and \( \text{rwork} \) is not nil, \( \text{rwork} \) contains the unconverged superdiagonal elements of an upper bidiagonal matrix. This is slightly different from LAPACK which puts unconverged superdiagonal elements in \( \text{work} \) if type is real; in \( \text{rwork} \) if type is complex. \( \text{rwork} \) can be a NULL pointer if the user does not want the information from superdiagonal.

Appendix G.1 provides a simple example of \( \text{gesvd} \).

Remark 1: \( \text{gesvd} \) only supports \( m \geq n \).

Remark 2: the routine returns \( V^H \), not \( V \).

API of \( \text{gesvd} \)

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobu</td>
<td>host</td>
<td>input</td>
<td>specifies options for computing all or part of the matrix $U$: = 'A': all m columns of $U$ are returned in array $U$; = 'S': the first min($m$, $n$) columns of $U$ (the left singular vectors) are returned in the array $U$; = 'O': the first min($m$, $n$) columns of $U$ (the left singular vectors) are overwitten on the array $A$; = 'N': no columns of $U$ (no left singular vectors) are computed.</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>jobvt</td>
<td>host</td>
<td>input</td>
<td>specifies options for computing all or part of the matrix $V^T$: = 'A': all N rows of $V^T$ are returned in the array $VT$; = 'S': the first min($m$, $n$) rows of $V^T$ (the right singular vectors) are returned in the array $VT$; = 'O': the first min($m$, $n$) rows of $V^T$ (the right singular vectors) are overwitten on the array $A$; = 'N': no rows of $V^T$ (no right singular vectors) are computed.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension $lda * n$ with $lda$ is not less than max(1, $m$). On exit, the contents of $A$ are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>S</td>
<td>device</td>
<td>output</td>
<td>real array of dimension min($m$, $n$). The singular values of $A$, sorted so that $S(i) &gt;= S(i+1)$.</td>
</tr>
<tr>
<td>U</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension $ldu * m$ with $ldu$ is not less than max(1, $m$). $U$ contains the $m \times m$ unitary matrix $U$.</td>
</tr>
<tr>
<td>ldu</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $U$.</td>
</tr>
<tr>
<td>VT</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension $ldvt * n$ with $ldvt$ is not less than max(1, $n$). $VT$ contains the $n \times n$ unitary matrix $V^T$.</td>
</tr>
<tr>
<td>ldvt</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $VT$.</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of work, returned by gesvd_bufferSize.</td>
</tr>
<tr>
<td>rwork</td>
<td>device</td>
<td>input</td>
<td>real array of dimension min($m$, $n$)-1. It contains the unconverged superdiagonal elements of an upper bidiagonal matrix if devInfo &gt; 0.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>devInfo</th>
<th>device</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>if devInfo = 0, the operation is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo &gt; 0, devInfo indicates how many superdiagonals of an intermediate bidiagonal form did not converge to zero.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (m, n&lt;0 or lda&lt;max(1,m) or ldu&lt;max(1,m) or ldvt&lt;max(1,n)).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

**2.4.3.7. cusolverDnGesvd() [DEPRECATED]**

[[DEPRECATED]] use cusolverDnXgesvd() instead. The routine will be removed in the next major release.

The helper functions below can calculate the sizes needed for pre-allocated buffer.

cusolverStatus_t cusolverDnGesvd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    signed char jobu,
    signed char jobvt,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeS,
    const void *S,
    cudaDataType dataTypeU,
    const void *U,
    int64_t ldu,
    cudaDataType dataTypeVT,
    const void *VT,
    int64_t ldvt,
    cudaDataType computeType,
    size_t *workspaceInBytes);

The routine below

cusolverStatus_t CUSOLVERAPI cusolverDnGesvd(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    signed char jobu,
    signed char jobvt,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
This function computes the singular value decomposition (SVD) of a $m \times n$ matrix $A$ and corresponding the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^H$$

where $\Sigma$ is an $m \times n$ matrix which is zero except for its $\min(m, n)$ diagonal elements, $U$ is an $m \times m$ unitary matrix, and $V$ is an $n \times n$ unitary matrix. The diagonal elements of $\Sigma$ are the singular values of $A$; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of $U$ and $V$ are the left and right singular vectors of $A$.

The user has to provide working space which is pointed by input parameter $pBuffer$. The input parameter $workspaceInBytes$ is size in bytes of the working space, and it is returned by $cusolverDnGesvd_bufferSize()$.

If output parameter $info = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle). If $bdsqr$ did not converge, $info$ specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.

Currently, $cusolverDnGesvd$ supports only the default algorithm.

**Table of algorithms supported by $cusolverDnGesvd$**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_ALG_0 or NULL</td>
<td>Default algorithm.</td>
</tr>
</tbody>
</table>

Appendix G.5 provides a simple example of $cusolverDnGesvd$.

**Remark 1:** $gesvd$ only supports $m \geq n$.

**Remark 2:** the routine returns $V^H$, not $V$.

List of input arguments for $cusolverDnGesvd_bufferSize$ and $cusolverDnGesvd$:

**API of $cusolverDnGesvd$**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by $cusolverDnSetAdvOptions$.</td>
</tr>
</tbody>
</table>
| jobu      | host   | input  | specifies options for computing all or part of the matrix $U$: = 'A': all $m$ columns of $U$ are returned in array $U$: = 'S': the first $\min(m,n)$ columns of $U$ [the left singular vectors] are returned in the array $U$: = 'O': the first $\min(m,n)$
columns of \( \mathbf{U} \) [the left singular vectors] are overwritten on the array \( \mathbf{A} \); \( \text{jobvt}' = \text{’N’} \): no columns of \( \mathbf{U} \) [no left singular vectors] are computed.

**jobvt** host input specifies options for computing all or part of the matrix \( \mathbf{V}^{**T} \): \( \text{jobvt}' = \text{’A’} \): all \( N \) rows of \( \mathbf{V}^{**T} \) are returned in the array \( \mathbf{VT} \); \( \text{jobvt}' = \text{’S’} \): the first \( \min(m,n) \) rows of \( \mathbf{V}^{**T} \) [the right singular vectors] are returned in the array \( \mathbf{VT} \); \( \text{jobvt}' = \text{’O’} \): the first \( \min(m,n) \) rows of \( \mathbf{V}^{**T} \) [the right singular vectors] are overwritten on the array \( \mathbf{A} \); \( \text{jobvt}' = \text{’N’} \): no rows of \( \mathbf{V}^{**T} \) [no right singular vectors] are computed.

<table>
<thead>
<tr>
<th>( \mathbf{m} )</th>
<th>host input</th>
<th>number of rows of matrix ( \mathbf{A} ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{n} )</td>
<td>host input</td>
<td>number of columns of matrix ( \mathbf{A} ).</td>
</tr>
<tr>
<td>( \text{dataTypeA} )</td>
<td>host input</td>
<td>data type of array ( \mathbf{A} ).</td>
</tr>
<tr>
<td>( \mathbf{A} )</td>
<td>device in/out</td>
<td>array of dimension ( \text{lda} \times n ) with ( \text{lda} ) is not less than ( \max(1, m) ). On exit, the contents of ( \mathbf{A} ) are destroyed.</td>
</tr>
<tr>
<td>( \text{lda} )</td>
<td>host input</td>
<td>leading dimension of two-dimensional array used to store matrix ( \mathbf{A} ).</td>
</tr>
<tr>
<td>( \text{dataTypeS} )</td>
<td>host input</td>
<td>data type of array ( \mathbf{S} ).</td>
</tr>
<tr>
<td>( \mathbf{S} )</td>
<td>device output</td>
<td>real array of dimension ( \min(m, n) ). The singular values of ( \mathbf{A} ), sorted so that ( S(i) \geq S(i+1) ).</td>
</tr>
<tr>
<td>( \text{dataTypeU} )</td>
<td>host input</td>
<td>data type of array ( \mathbf{U} ).</td>
</tr>
<tr>
<td>( \mathbf{U} )</td>
<td>device output</td>
<td>array of dimension ( \text{ldu} \times m ) with ( \text{ldu} ) is not less than ( \max(1, m) ). ( \mathbf{U} ) contains the ( m \times m ) unitary matrix ( \mathbf{U} ).</td>
</tr>
<tr>
<td>( \text{ldu} )</td>
<td>host input</td>
<td>leading dimension of two-dimensional array used to store matrix ( \mathbf{U} ).</td>
</tr>
<tr>
<td>( \text{dataTypeVT} )</td>
<td>host input</td>
<td>data type of array ( \mathbf{VT} ).</td>
</tr>
<tr>
<td>( \mathbf{VT} )</td>
<td>device output</td>
<td>array of dimension ( \text{ldvt} \times n ) with ( \text{ldvt} ) is not less than ( \max(1, n) ). ( \mathbf{VT} ) contains the ( n \times n ) unitary matrix ( \mathbf{V}^{**T} ).</td>
</tr>
<tr>
<td>( \text{ldvt} )</td>
<td>host input</td>
<td>leading dimension of two-dimensional array used to store matrix ( \mathbf{VT} ).</td>
</tr>
<tr>
<td>( \text{computeType} )</td>
<td>host input</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>( \text{pBuffer} )</td>
<td>device in/out</td>
<td>Working space. Array of type void of size workspaceInBytes bytes.</td>
</tr>
<tr>
<td>( \text{workspaceInBytes} )</td>
<td>host input</td>
<td>Size in bytes of ( \text{pBuffer} ), returned by ( \text{cusolverDnGesvd_bufferSize} ).</td>
</tr>
<tr>
<td>( \text{info} )</td>
<td>device output</td>
<td>if ( \text{info} = 0 ), the operation is successful. if ( \text{info} = -i ), the ( i )-th parameter is wrong (not counting handle). if ( \text{info} &gt; 0 ), ( \text{info} ) indicates how many superdiagonals</td>
</tr>
</tbody>
</table>
of an intermediate bidiagonal form did not converge to zero.

The generic API has three different types, data type of the matrix A, data type of the vector S and data type of the matrix U, data type of the matrix VT, compute type is compute type of the operation. cusolverDnGesvd only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>data type A</th>
<th>data type S</th>
<th>data type U</th>
<th>data type VT</th>
<th>compute type</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGESVD</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGESVD</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGESVD</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGESVD</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.3.8. cusolverDn<t>gesvdj()###

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverDnSgesvdj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    int m,
    int n,
    const float *A,
    int lda,
    const float *S,
    const float *U,
    int ldu,
    const float *V,
    int ldv,
    int *lwork,
    gsvdjInfo_t params);
```

```c
cusolverStatus_t
cusolverDnDgesvdj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    int m,
    int n,
    const double *A,
    int lda,
    const double *S,
    const double *U,
    int ldu,
```
The S and D data types are real valued single and double precision, respectively.
The C and Z data types are complex valued single and double precision, respectively.

This function computes the singular value decomposition (SVD) of a $m \times n$ matrix $A$ and corresponding the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^H$$

where $\Sigma$ is an $m \times n$ matrix which is zero except for its $\min(m, n)$ diagonal elements, $U$ is an $m \times m$ unitary matrix, and $V$ is an $n \times n$ unitary matrix. The diagonal elements of $\Sigma$ are the singular

```c
double *A,
int lda,
double *S,
double *U,
int ldu,
double *V,
int ldv,
double *work,
int lwork,
int *info,
gesvdjInfo_t params);
```

cusolverDnCgesvdj(
    cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
    int econ,
    int m,
    int n,
cuComplex *A,
    int lda,
    float *S,
cuComplex *U,
    int ldu,
cuComplex *V,
    int ldv,
cuComplex *work,
    int lwork,
    int *info,
gesvdjInfo_t params);

```c

cusolverDnZgesvdj(
    cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
    int econ,
    int m,
    int n,
cuDoubleComplex *A,
    int lda,
cuDoubleComplex *S,
cuDoubleComplex *U,
    int ldu,
cuDoubleComplex *V,
    int ldv,
cuDoubleComplex *work,
    int lwork,
    int *info,
gesvdjInfo_t params);
```
values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m, n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

gesvdj has the same functionality as gesvd. The difference is that gesvd uses QR algorithm and gesvdj uses Jacobi method. The parallelism of Jacobi method gives GPU better performance on small and medium size matrices. Moreover the user can configure gesvdj to perform approximation up to certain accuracy.

gesvdj iteratively generates a sequence of unitary matrices to transform matrix \( A \) to the following form

\[
U^H A V = S + E
\]

where \( S \) is diagonal and diagonal of \( E \) is zero.

During the iterations, the Frobenius norm of \( E \) decreases monotonically. As \( E \) goes down to zero, \( S \) is the set of singular values. In practice, Jacobi method stops if

\[
||E||_F \leq \text{eps} \, ||A||_F
\]

where \( \text{eps} \) is given tolerance.

gesvdj has two parameters to control the accuracy. First parameter is tolerance (\( \text{eps} \)). The default value is machine accuracy but the user can use function cuSolverDnXgesvdjSetTolerance to set a priori tolerance. The second parameter is maximum number of sweeps which controls number of iterations of Jacobi method. The default value is 100 but the user can use function cuSolverDnXgesvdjSetMaxSweeps to set a proper bound. The experiments show 15 sweeps are good enough to converge to machine accuracy. gesvdj stops either tolerance is met or maximum number of sweeps is met.

Jacobi method has quadratic convergence, so the accuracy is not proportional to number of sweeps. To guarantee certain accuracy, the user should configure tolerance only.

The user has to provide working space which is pointed by input parameter \( \text{work} \). The input parameter \( \text{lwork} \) is the size of the working space, and it is returned by gesvdj_bufferSize().

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle). If \( \text{info} = \min(m, n)+1 \), gesvdj does not converge under given tolerance and maximum sweeps.

If the user sets an improper tolerance, gesvdj may not converge. For example, tolerance should not be smaller than machine accuracy.

Appendix G.2 provides a simple example of gesvdj.

Remark 1: gesvdj supports any combination of \( m \) and \( n \).

Remark 2: the routine returns \( V \), not \( V^H \). This is different from gesvd.

**API of gesvdj**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute singular value only or</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

CUSOLVER Library

DU-06709-001_v11.1   |   138

singular vectors as well: jobz = CUSOLVER_EIG_MODE_NOVECTOR:
Compute singular values only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute
singular values and singular vectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>econ</td>
<td>host</td>
<td>input</td>
<td>econ = 1 for economy size for U and V.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n with lda is not less than max(1,m). On exit, the contents of A are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>S</td>
<td>device</td>
<td>output</td>
<td>real array of dimension min(m,n). The singular values of A, sorted so that S(i) &gt;= S(i+1).</td>
</tr>
<tr>
<td>U</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ldu * m if econ is zero. If econ is nonzero, the dimension is ldu * min(m,n). U contains the left singular vectors.</td>
</tr>
<tr>
<td>ldu</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix U. ldu is not less than max(1,m).</td>
</tr>
<tr>
<td>V</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ldv * n if econ is zero. If econ is nonzero, the dimension is ldv * min(m,n). V contains the right singular vectors.</td>
</tr>
<tr>
<td>ldv</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix V. ldv is not less than max(1,n).</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of size lwork, working space.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of work, returned by gesvdj_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = min(m,n)+1, gesvdj dose not converge under given tolerance and maximum sweeps.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>in/out</td>
<td>structure filled with parameters of Jacobi algorithm and results of gesvdj.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_INVALID_VALUE
invalid parameters were passed \( m, n < 0 \)
or \( lda < \max(1, m) \) or \( ldu < \max(1, m) \)
or \( ldv < \max(1, n) \) or \( jobz \) is not
CUSOLVER_EIG_MODE_NOVECTOR or
CUSOLVER_EIG_MODE_VECTOR.

CUSOLVER_STATUS_INTERNAL_ERROR
an internal operation failed.

2.4.3.9. cusolverDn<t>gesvdjBatched()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

cusolverStatus_t
cusolverDnSgesvdjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int m,
    int n,
    const float *A,
    int lda,
    const float *S,
    const float *U,
    int ldu,
    const float *V,
    int ldv,
    int *lwork,
    gesvdjInfo_t params,
    int batchSize);

cusolverStatus_t
cusolverDnDgesvdjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int m,
    int n,
    const double *A,
    int lda,
    const double *S,
    const double *U,
    int ldu,
    const double *V,
    int ldv,
    int *lwork,
    gesvdjInfo_t params,
    int batchSize);

cusolverStatus_t
cusolverDnCgesvdjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int m,
    int n,
    const cuComplex *A,
    int lda,
    const float *S,
    const cuComplex *U,
    int ldu,
    const cuComplex *V,
    int ldv,
    int *lwork,
    gesvdjInfo_t params,
    int batchSize);

cusolverStatus_t
cusolverDnZgesvdjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int m,
    int n,
    const cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *S,
    const cuDoubleComplex *U,
    int ldu,
    const cuDoubleComplex *V,
    int ldv,
    int *lwork,
    gesvdjInfo_t params,
    int batchSize);
Using the CUSOLVER API

```c
 cusolverDnHandle_t handle,
 cusolverEligMode_t jobz,
 int m,
 int n,
 const cuDoubleComplex *A,
 int lda,
 const double *S,
 const cuDoubleComplex *U,
 int ldu,
 const cuDoubleComplex *V,
 int ldv,
 int *lwork,
 gesvdjInfo_t params,
 int batchSize);
```

The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
 cusolverDnSgesvdjBatched(
   cusolverDnHandle_t handle,
   cusolverEligMode_t jobz,
   int m,
   int n,
   float *A,
   int lda,
   float *S,
   float *U,
   int ldu,
   float *V,
   int ldv,
   float *work,
   int lwork,
   int *info,
   gesvdjInfo_t params,
   int batchSize);)
```

```c
 cusolverStatus_t
 cusolverDnDgesvdjBatched(
   cusolverDnHandle_t handle,
   cusolverEligMode_t jobz,
   int m,
   int n,
   double *A,
   int lda,
   double *S,
   double *U,
   int ldu,
   double *V,
   int ldv,
   double *work,
   int lwork,
   int *info,
   gesvdjInfo_t params,
   int batchSize);)
```

The C and Z data types are complex valued single and double precision, respectively.

```c
 cusolverStatus_t
 cusolverDnCgesvdjBatched(
   cusolverDnHandle_t handle,
   cusolverEligMode_t jobz,
   int m,
   int n,
   const cuDoubleComplex *A,
   int lda,
   const double *S,
   const cuDoubleComplex *U,
   int ldu,
   const cuDoubleComplex *V,
   int ldv,
   int *lwork,
   gesvdjInfo_t params,
   int batchSize);
```
This function computes singular values and singular vectors of a sequence of general $m \times n$ matrices

$$A_j = U_j \Sigma_j V_j^H$$

where $\Sigma_j$ is a real $m \times n$ diagonal matrix which is zero except for its $\min(m, n)$ diagonal elements. $U_j$ (left singular vectors) is a $m \times m$ unitary matrix and $V_j$ (right singular vectors) is a $n \times n$ unitary matrix. The diagonal elements of $\Sigma_j$ are the singular values of $A_j$ in either descending order or non-sorting order.

gesvdjBatched performs gesvdj on each matrix. It requires that all matrices are of the same size $m, n$ no greater than 32 and are packed in contiguous way,

$$A = (A_0 \ A_1 \ \ldots)$$

Each matrix is column-major with leading dimension lda, so the formula for random access is $A_k(i, j) = A[i + lda* j + lda*n*k]$.

The parameter $S$ also contains singular values of each matrix in contiguous way,

$$S = (S_0 \ S_1 \ \ldots)$$

The formula for random access of $S$ is $S_k(j) = S[j + \min(m, n)*k]$. 
Except for tolerance and maximum sweeps, gesvdjBatched can either sort the singular values in descending order (default) or chose as-is [without sorting] by the function cusolverDnXgesvdjSetSortEig. If the user packs several tiny matrices into diagonal blocks of one matrix, non-sorting option can separate singular values of those tiny matrices.

gesvdjBatched cannot report residual and executed sweeps by function cusolverDnXgesvdjGetResidual and cusolverDnXgesvdjGetSweeps. Any call of the above two returns CUSOLVER_STATUS_NOT_SUPPORTED. The user needs to compute residual explicitly.

The user has to provide working space pointed by input parameter work. The input parameter lwork is the size of the working space, and it is returned by gesvdjBatched_bufferSize().

The output parameter info is an integer array of size batchSize. If the function returns CUSOLVER_STATUS_INVALID_VALUE, the first element info[0] = -i (less than zero) indicates i-th parameter is wrong (not counting handle). Otherwise, if info[i] = min(m,n)+1, gesvdjBatched does not converge on i-th matrix under given tolerance and maximum sweeps.

Appendix G.3 provides a simple example of gesvdjBatched.

API of syevjBatched

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute singular value only or singular vectors as well: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute singular values only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute singular values and singular vectors.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix Aj. m is no greater than 32.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix Aj. n is no greater than 32.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n * batchSize with lda is not less than max(1,n). on Exit: the contents of Aj are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix Aj.</td>
</tr>
<tr>
<td>S</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension min(m,n)*batchSize. It stores the singular values of Aj in descending order or non-sorting order.</td>
</tr>
<tr>
<td>U</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension ldv * m * batchSize. Uj contains the left singular vectors of Aj.</td>
</tr>
</tbody>
</table>
### Using the CUSOLVER API

**ldu**  
**host**  
**input**  
leading dimension of two-dimensional array used to store matrix \( U_j \). \( ldu \) is not less than \( \max(1, m) \).

**V**  
**device**  
**output**  
\(<\text{type}>\) array of dimension \( ldv \times n \times batchSize \). \( V_j \) contains the right singular vectors of \( A_j \).

**ldv**  
**host**  
**input**  
leading dimension of two-dimensional array used to store matrix \( V_j \). \( ldv \) is not less than \( \max(1, n) \).

**work**  
**device**  
**in/out**  
\(<\text{type}>\) array of size \( lwork \), working space.

**lwork**  
**host**  
**input**  
size of \( work \), returned by \( gesvdjBatched_bufferSize \).

**info**  
**device**  
**output**  
an integer array of dimension \( batchSize \). If \( \text{CUSOLVER_STATUS_INVALID_VALUE} \) is returned, \( \text{info}[0] = -i \) (less than zero) indicates \( i\)-th parameter is wrong (not counting handle). Otherwise, if \( \text{info}[i] = 0 \), the operation is successful. If \( \text{info}[i] = \min(m, n)+1 \), \( \text{gesvdjBatched} \) does not converge on \( i\)-th matrix under given tolerance and maximum sweeps.

**params**  
**host**  
**in/out**  
structure filled with parameters of Jacobi algorithm.

**batchSize**  
**host**  
**input**  
number of matrices.

### Status Returned

<table>
<thead>
<tr>
<th>Status Return</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( m, n&lt;0 ) or ( lda&lt;\max(1, m) ) or ( ldu&lt;\max(1, m) ) or ( ldv&lt;\max(1, n) ) or ( \text{jobz} ) is not ( \text{CUSOLVER_EIG_MODE_NOVECTOR} ) or ( \text{CUSOLVER_EIG_MODE_VECTOR} ), or ( \text{batchSize}&lt;0 ).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.3.10. \texttt{cusolverDn\_t\_gesvdaStridedBatched()}\texttt{(\ldots)}

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c

cusolverStatus_t
cusolverDnSgesvdaStridedBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int rank,
    int m,
    int n,
```
const float *A,
  int lda,
  long long int strideA,
const float *S,
  long long int strideS,
const float *U,
  int ldu,
  long long int strideU,
const float *V,
  int ldv,
  long long int strideV,
  int *lwork,
  int batchSize);

cusolverStatus_t
cusolverDnDgesvdaStridedBatched_bufferSize(
  cusolverDnHandle_t handle,
  cusolverEigMode_t jobz,
  int rank,
  int m,
  int n,
const double *A,
  int lda,
  long long int strideA,
const double *S,
  long long int strideS,
const double *U,
  int ldu,
  long long int strideU,
const double *V,
  int ldv,
  long long int strideV,
  int *lwork,
  int batchSize);

cusolverStatus_t
cusolverDnCgesvdaStridedBatched_bufferSize(
  cusolverDnHandle_t handle,
  cusolverEigMode_t jobz,
  int rank,
  int m,
  int n,
const cuComplex *A,
  int lda,
  long long int strideA,
const double *S,
  long long int strideS,
const cuComplex *U,
  int ldu,
  long long int strideU,
const cuComplex *V,
  int ldv,
  long long int strideV,
  int *lwork,
  int batchSize);

cusolverStatus_t
cusolverDnZgesvdaStridedBatched_bufferSize(
  cusolverDnHandle_t handle,
  cusolverEigMode_t jobz,
  int rank,
  int m,
  int n,
const cuDoubleComplex *A,
  int lda,
  long long int strideA,
const double *S,
  long long int strideS,
const cuDoubleComplex *U,
int ldu,
long long int strideU,
const cuDoubleComplex *V,
int ldv,
long long int strideV,
int *lwork,
int batchSize);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSgesvdaStridedBatched(    
cusolverDnHandle_t handle,    
cusolverEigMode_t jobz,    
int rank,    
int m,    
int n,    
const float *A,    
int lda,    
long long int strideA,    
float *S,    
long long int strideS,    
float *U,    
int ldu,    
long long int strideU,    
float *V,    
int ldv,    
long long int strideV,    
float *work,    
int lwork,    
int *info,    
double *h_R_nrmF,    
int batchSize);

cusolverStatus_t
cusolverDnDgesvdaStridedBatched(    
cusolverDnHandle_t handle,    
cusolverEigMode_t jobz,    
int rank,    
int m,    
int n,    
const double *A,    
int lda,    
long long int strideA,    
double *S,    
long long int strideS,    
double *U,    
int ldu,    
long long int strideU,    
double *V,    
int ldv,    
long long int strideV,    
double *work,    
int lwork,    
int *info,    
double *h_R_nrmF,    
int batchSize);

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnCgesvdaStridedBatched(    
cusolverDnHandle_t handle,    
cusolverEigMode_t jobz,
This function `gesvda` (a stands for approximate) approximates the singular value decomposition of a tall skinny $m \times n$ matrix $A$ and corresponding the left and right singular vectors. The economy form of SVD is written by

$$A = U \Sigma V^H$$

where $\Sigma$ is an $n \times n$ matrix, $U$ is an $m \times n$ unitary matrix, and $V$ is an $n \times n$ unitary matrix. The diagonal elements of $\Sigma$ are the singular values of $A$; they are real and non-negative, and are returned in descending order. $U$ and $V$ are the left and right singular vectors of $A$.

gesvda computes eigenvalues of $A^T A$ to approximate singular values and singular vectors. It generates matrices $U$ and $V$ and transforms the matrix $A$ to the following form

$$U^H A V = S + E$$
where $S$ is diagonal and $E$ depends on rounding errors. To certain conditions, $U$, $V$ and $S$
approximate singular values and singular vectors up to machine zero of single precision. In
general, $V$ is unitary, $S$ is more accurate than $U$. If singular value is far from zero, then left
singular vector $U$ is accurate. In other words, the accuracy of singular values and left singular
vectors depend on the distance between singular value and zero.

The input parameter `rank` decides the number of singular values and singular vectors are
computed in parameter $S$, $U$ and $V$.

The output parameter `h_RnrmF` computes Frobenius norm of residual.

$$A - U \cdot S \cdot V^H$$

if the parameter `rank` is equal $n$. Otherwise, `h_RnrmF` reports

$$||U \cdot S \cdot V^H|| - ||S||$$
in Frobenius norm sense. That is, how far $U$ is from unitary.

`gesvdaStridedBatched` performs `gesvda` on each matrix. It requires that all matrices are of
the same size $m$, $n$ and are packed in contiguous way,

$$A = (A_0 \ A_1 \ \cdots)$$

Each matrix is column-major with leading dimension $lda$, so the formula for random
access is $A_k(i,j) = A[i + lda^*j] + strideA^*k$. Similarly, the formula for random
access of $S$ is $S_k(i,j) = S[j + strideS^*k]$, the formula for random access of $U$ is
$U_k(i,j) = U[i + ldv^*j + strideU^*k]$ and the formula for random access of $V$ is
$V_k(i,j) = V[i + ldv^*j + strideV^*k]$.

The user has to provide working space which is pointed by input parameter `work`.
The input parameter `lwork` is the size of the working space, and it is returned by `gesvdaStridedBatched_bufferSize()`.

The output parameter `info` is an integer array of size `batchSize`. If the function returns
`CUSOLVER_STATUS_INVALID_VALUE`, the first element `info[0] = -i` (less than zero)
indicates $i$-th parameter is wrong (not counting handle). Otherwise, if `info[i] = \min(m, n) + 1`, `gesvdaStridedBatched` does not converge on $i$-th matrix under given
tolerance.

Appendix G.4 provides a simple example of `gesvda`.

Remark 1: the routine returns $V$, not $V^H$. This is different from `gesvd`.

Remark 2: if the user is confident on the accuracy of singular values and singular vectors,
for example, certain conditions hold (required singular value is far from zero), then the
performance can be improved by passing null pointer to `h_RnrmF`, i.e. no computation of
residual norm.

API of `gesvda`

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute singular value only or</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

**singular vectors as well:**

```
jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute singular values only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute singular values and singular vectors.
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rank</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>m</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>A</code></td>
<td>device</td>
<td>input</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>strideA</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>S</code></td>
<td>device</td>
<td>output</td>
</tr>
<tr>
<td><code>strideS</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>U</code></td>
<td>device</td>
<td>output</td>
</tr>
<tr>
<td><code>ldu</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>strideU</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>V</code></td>
<td>device</td>
<td>output</td>
</tr>
<tr>
<td><code>ldv</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>strideV</code></td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td><code>work</code></td>
<td>device</td>
<td>in/out</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>lwork</th>
<th>host</th>
<th>input</th>
<th>size of work, returned by gesvdaStridedBatched_bufferSize.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>an integer array of dimension batchSize. If CUSOLVER_STATUS_INVALID_VALUE is returned, info[0] = -i (less than zero) indicates i-th parameter is wrong (not counting handle). Otherwise, if info[i] = 0, the operation is successful. if info[i] = min(m,n)+1, gesvdaStridedBatched dose not converge on i-th matrix.</td>
</tr>
<tr>
<td>h_RnrmF</td>
<td>host</td>
<td>output</td>
<td>&lt;double&gt; array of size batchSize. h_RnrmF[i] is norm of residual of i-th matrix.</td>
</tr>
<tr>
<td>batchSize</td>
<td>host</td>
<td>input</td>
<td>number of matrices. batchSize is not less than 1.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( m, n &lt; 0 ) or ( \text{lda} &lt; \max(1, m) ) or ( \text{ldu} &lt; \max(1, m) ) or ( \text{ldv} &lt; \max(1, n) ) or ( \text{strideA} &lt; \text{lda} \times n ) or ( \text{strideS} &lt; \text{rank} ) or ( \text{strideU} &lt; \text{ldu} \times \text{rank} ) or ( \text{strideV} &lt; \text{ldv} \times \text{rank} ) or ( \text{batchSize} &lt; 1 ) or ( \text{jobz} ) is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.3.11. cusolverDn<t>syevd()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

cusolverStatus_t
cusolverDnSyevd_bufferSize(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
const float *A,
int lda,
const float *W,
int *lwork);

cusolverStatus_t
cusolverDnDsyevd_bufferSize(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
const double *A,
int lda,
const double *W,
int *lwork);
Using the CUSOLVER API

```c
cusolverStatus_t
cusolverDnCheevd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuComplex *A,
    int lda,
    const float *W,
    int *lwork);

cusolverStatus_t
cusolverDnZheevd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuDoubleComplex *A,
    int lda,
    const double *W,
    int *lwork);
```

The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnSsyevd(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    float *W,
    float *work,
    int lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnDsyevd(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *W,
    double *work,
    int lwork,
    int *devInfo);
```

The C and Z data types are complex valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnCheevd(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    float *W,
    cuComplex *work,
    int lwork,
    int *devInfo);
```
cusolverStatus_t
cusolverDnZheevd(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *devInfo);

This function computes eigenvalues and eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix \( A \). The standard symmetric eigenvalue problem is

\[
A V = V \Lambda
\]

where \( \Lambda \) is a real \( n \times n \) diagonal matrix. \( V \) is an \( n \times n \) unitary matrix. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

The user has to provide working space which is pointed by input parameter work. The input parameter lwork is size of the working space, and it is returned by \texttt{syevd_bufferSize()}.

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle). If devInfo = i (greater than zero), i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If \( \text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR} \), \( A \) contains the orthonormal eigenvectors of the matrix \( A \). The eigenvectors are computed by a divide and conquer algorithm.

Appendix F.1 provides a simple example of \texttt{syevd}.

### API of \texttt{syevd}

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: ( \text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR} ) : Compute eigenvalues only; ( \text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR} ) : Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of ( A ) is stored. ( \text{uplo} = \text{CUBLAS_FILL_MODE_LOWER} ) : Lower triangle of ( A ) is stored. ( \text{uplo} = \text{CUBLAS_FILL_MODE_UPPER} ) : Upper triangle of ( A ) is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows [or columns] of matrix ( A ).</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) array of dimension ( \text{lda} \times n ) with ( \text{lda} ) is not less than ( \max(1, n) ). If ( \text{uplo} = \text{CUBLAS_FILL_MODE_UPPER} ), the leading ( n )-by-( n ) upper triangular part of ( A ) contains the upper triangular</td>
</tr>
</tbody>
</table>
part of the matrix $A$. If $\text{uplo} = \text{CUBLAS_FILL_MODE_LOWER}$, the leading $n$-by-$n$ lower triangular part of $A$ contains the lower triangular part of the matrix $A$. On exit, if $\text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR}$, and $\text{devInfo} = 0$, $A$ contains the orthonormal eigenvectors of the matrix $A$. If $\text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR}$, the contents of $A$ are destroyed.

<table>
<thead>
<tr>
<th>lda</th>
<th>host</th>
<th>input</th>
<th>leading dimension of two-dimensional array used to store matrix $A$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W$</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension $n$. The eigenvalue values of $A$, in ascending order ie, sorted so that $W(i) \leq W(i+1)$.</td>
</tr>
<tr>
<td>$\text{work}$</td>
<td>device</td>
<td>in/out</td>
<td>working space, $\text{&lt;type&gt;} \text{ array of size } l\text{work}$.</td>
</tr>
<tr>
<td>$l\text{work}$</td>
<td>host</td>
<td>input</td>
<td>size of $\text{work}$, returned by $\text{syevd_bufferSize}$.</td>
</tr>
<tr>
<td>$\text{devInfo}$</td>
<td>device</td>
<td>output</td>
<td>if $\text{devInfo} = 0$, the operation is successful. if $\text{devInfo} = -i$, the $i$-th parameter is wrong (not counting $\text{handle}$). if $\text{devInfo} = i$ ($&gt;0$), $\text{devInfo}$ indicates $i$ off-diagonal elements of an intermediate tridiagonal form did not converge to zero;</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**: invalid parameters were passed ($n<0$, or $l\text{da}<\text{max}(1, n)$, or $\text{jobz}$ is not $\text{CUSOLVER_EIG_MODE_NOVECTOR}$ or $\text{CUSOLVER_EIG_MODE_VECTOR}$, or $\text{uplo}$ is not $\text{CUBLAS_FILL_MODE_LOWER}$ or $\text{CUBLAS_FILL_MODE_UPPER}$).
- **CUSOLVER_STATUS_ARCH_MISMATCH**: the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.

**2.4.3.12. cusolverDnSyevd()[DEPRECATED]**

[[DEPRECATED]] use $\text{cusolverDnXsyevd}()$ instead. The routine will be removed in the next major release.

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t cusolverDnSyevd_bufferSize(
    cusolverDnHandle_t handle,
```
The routine below

```c
 cusolverDnSyevd(
    cusolverDnHandle_t handle, 
    cusolverParams_t params, 
    cusolverEigMode_t jobz, 
    cublasFillMode_t uplo, 
    int n, 
    cudaDataType dataTypeA, 
    const void *A, 
    int64_t lda, 
    cudaDataType dataTypeW, 
    const void *W, 
    cudaDataType computeType, 
    void *pBuffer, 
    size_t workspaceInBytes, 
    int *info);
```

computes eigenvalues and eigenvectors of a symmetric (Hermitian) $n \times n$ matrix $A$ using the generic API interface. The standard symmetric eigenvalue problem is

$A V = V \Lambda$

where $\Lambda$ is a real $n \times n$ diagonal matrix. $V$ is an $n \times n$ unitary matrix. The diagonal elements of $\Lambda$ are the eigenvalues of $A$ in ascending order.

The user has to provide working space which is pointed by input parameter pBuffer. The input parameter workspaceInBytes is size in bytes of the working space, and it is returned by cusolverDnSyevd_bufferSize().

If output parameter info = -i (less than zero), the i-th parameter is wrong (not counting handle). If info = i (greater than zero), i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If jobz = CUSOLVER_EIG_MODE_VECTOR, $A$ contains the orthonormal eigenvectors of the matrix $A$. The eigenvectors are computed by a divide and conquer algorithm.

Currently, cusolverDnSyevd supports only the default algorithm.

**Table of algorithms supported by cusolverDnSyevd**

<table>
<thead>
<tr>
<th>CUSOLVER_ALG_0 or NULL</th>
<th>Default algorithm.</th>
</tr>
</thead>
</table>

**List of input arguments for cusolverDnSyevd_bufferSize and cusolverDnSyevd:**

**API of cusolverDnSyevd**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
</tbody>
</table>
### Using the CUSOLVER API

<table>
<thead>
<tr>
<th>params</th>
<th>host</th>
<th>input</th>
<th>structure with information collected by cusolverDnSetAdvOptions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of A is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension lda * n with lda is not less than max(1,n). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A. On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info = 0, A contains the orthonormal eigenvectors of the matrix A. If jobz = CUSOLVER_EIG_MODE_NOVECTOR, the contents of A are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>dataTypeW</td>
<td>host</td>
<td>in</td>
<td>data type of array W.</td>
</tr>
<tr>
<td>W</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension n. The eigenvalue values of A, in ascending order ie, sorted so that W(i) &lt;= W(i+1).</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>in/out</td>
<td>Working space. Array of type void of size workspaceInBytes bytes.</td>
</tr>
<tr>
<td>workspaceInBytes</td>
<td>host</td>
<td>input</td>
<td>Size in bytes of pBuffer, returned by cusolverDnSyevd_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i (&gt; 0), info indicates i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;</td>
</tr>
</tbody>
</table>
The generic API has three different types, `dataTypeA` is data type of the matrix $A$, `dataTypeW` is data type of the matrix $W$ and `computeType` is compute type of the operation. `cusolverDnSyevd` only supports the following four combinations.

### Valid combination of data type and compute type

<table>
<thead>
<tr>
<th><code>dataTypeA</code></th>
<th><code>dataTypeW</code></th>
<th><code>computeType</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SSYEVD</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DSYEVD</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CHEEVD</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZHEEVD</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ($n&lt;0$, or $lda&lt;\max(1, n)$, or <code>jobz</code> is not <code>CUSOLVER_EIG_MODE_NOVECTOR</code> or <code>CUSOLVER_EIG_MODE_VECTOR</code>, or <code>uplo</code> is not <code>CUBLAS_FILL_MODE_LOWER</code> or <code>CUBLAS_FILL_MODE_UPPER</code>).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

#### 2.4.3.13. `cusolverDn<t>syevdx()`

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t cusolverDnSyevdx_bufferSize(   
    cusolverDnHandle_t handle,   
    cusolverEigMode_t jobz,   
    cusolverEigRange_t range,   
    cublasFillMode_t uplo,   
    int n,   
    const float *A,   
    int lda,   
    float vl,   
    float vu,   
    int il,   
    int iu,   
    int *h_meig,   
    const float *W,   
    int *lwork);
```

```c
cusolverStatus_t cusolverDnDsyevdx_bufferSize(   
    cusolverDnHandle_t handle,   
    cusolverEigMode_t jobz,   
    cusolverEigRange_t range,   
    cublasFillMode_t uplo,   
    int n,   
    const double *A,   
    int lda,   
    double vl,   
    double vu,   
    int il,   
    int iu,   
    int *h_meig,   
    const double *W,   
    int *lwork);
```
Using the CUSOLVER API

double vu, 
int il, 
int iu, 
int *h_meig, 
const double *W, 
int *lwork);

cusolverStatus_t
cusolverDnCheevdx_bufferSize(
cusolverDnHandle_t handle, 
cusolverEigMode_t jobz, 
cusolverEigRange_t range, 
cublasFillMode_t uplo, 
int n, 
const cuComplex *A, 
int lda, 
float vl, 
float vu, 
int il, 
int iu, 
int *h_meig, 
const float *W, 
int *lwork);

cusolverStatus_t
cusolverDnZheevdx_bufferSize(
cusolverDnHandle_t handle, 
cusolverEigMode_t jobz, 
cusolverEigRange_t range, 
cublasFillMode_t uplo, 
int n, 
const cuDoubleComplex *A, 
int lda, 
double vl, 
double vu, 
int il, 
int iu, 
int *h_meig, 
const double *W, 
int *lwork);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSsyevdx(
cusolverDnHandle_t handle, 
cusolverEigMode_t jobz, 
cusolverEigRange_t range, 
cublasFillMode_t uplo, 
int n, 
float *A, 
int lda, 
float vl, 
float vu, 
int il, 
int iu, 
int *h_meig, 
float *W, 
float *work, 
int lwork, 
int *devInfo);

cusolverStatus_t
cusolverDnDsyevdx(
cusolverDnHandle_t handle, 
cusolverEigMode_t jobz, 
cusolverEigRange_t range, 
cublasFillMode_t uplo, 
int n, 
float *A, 
int lda, 
float vl, 
float vu, 
int il, 
int iu, 
int *h_meig, 
float *W, 
float *work, 
int lwork, 
int *devInfo);
The C and Z data types are complex valued single and double precision, respectively.

```c
int n,
double *A,
int lda,
double vl,
double vu,
int il,
int iu,
int *h_meig,
double *W,
double *work,
int lwork,
int *devInfo);
```

This function computes all or selection of the eigenvalues and optionally eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix \( A \). The standard symmetric eigenvalue problem is

\[
A V = V \Lambda
\]

where \( \Lambda \) is a real \( n \times h_{\text{meig}} \) diagonal matrix, \( V \) is an \( n \times h_{\text{meig}} \) unitary matrix, \( h_{\text{meig}} \) is the number of eigenvalues/eigenvectors computed by the routine, \( h_{\text{meig}} \) is equal to \( n \) when
the whole spectrum (e.g., range = CUSOLVER_EIG_RANGE_ALL) is requested. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

The user has to provide working space which is pointed by input parameter work. The input parameter lwork is size of the working space, and it is returned by syevdx_bufferSize().

If output parameter devInfo = -i (less than zero), the i-th parameter is wrong (not counting handle). If devInfo = i (greater than zero), i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If jobz = CUSOLVER_EIG_MODE_VECTOR, \( A \) contains the orthonormal eigenvectors of the matrix \( A \). The eigenvectors are computed by a divide and conquer algorithm.

Appendix F.1 provides a simple example of syevdx.

**API of syevdx**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>range</td>
<td>host</td>
<td>input</td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: range = CUSOLVER_EIG_RANGE_ALL: all eigenvalues/eigenvectors will be found, will become the classical syevd/heevd routine; range = CUSOLVER_EIG_RANGE_V: all eigenvalues/eigenvectors in the half-open interval [vl,vu] will be found; range = CUSOLVER_EIG_RANGE_I: the il-th through iu-th eigenvalues/eigenvectors will be found;</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of ( A ) is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of ( A ) is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of ( A ) is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows [or columns] of matrix ( A ).</td>
</tr>
<tr>
<td>( A )</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n with lda is not less than ( \max(1,n) ). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of ( A ) contains the upper triangular part of the matrix ( A ). If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of ( A ) contains the lower triangular part of ( A ).</td>
</tr>
</tbody>
</table>
part of the matrix $A$. On exit, if $\text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR}$, and $\text{devInfo} = 0$, $A$ contains the orthonormal eigenvectors of the matrix $A$. If $\text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR}$, the contents of $A$ are destroyed.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{lda}$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$. $\text{lda}$ is not less than $\max(1,n)$.</td>
</tr>
<tr>
<td>$\text{vl, vu}$</td>
<td>host</td>
<td>input</td>
<td>real values float or double for $[C, S]$ or $[Z, D]$ precision respectively. If $\text{range} = \text{CUSOLVER_EIG_RANGE_V}$, the lower and upper bounds of the interval to be searched for eigenvalues. $\text{vl} &gt; \text{vu}$. Not referenced if $\text{range} = \text{CUSOLVER_EIG_RANGE_ALL} \text{ or } \text{range} = \text{CUSOLVER_EIG_RANGE_I}$. Note that, if eigenvalues are very close to each other, it is well known that two different eigenvalues routines might find slightly different number of eigenvalues inside the same interval. This is due to the fact that different eigenvalue algorithms, or even same algorithm but different run might find eigenvalues within some rounding error close to the machine precision. Thus, if the user want to be sure not to miss any eigenvalue within the interval bound, we suggest that, the user substract/add epsilon (machine precision) to the interval bound such as $[\text{vl} = \text{vl}-\epsilon, \text{vu} = \text{vu}+\epsilon]$. This suggestion is valid for any selective routine from cuSolver or LAPACK.</td>
</tr>
<tr>
<td>$\text{il, iu}$</td>
<td>host</td>
<td>input</td>
<td>integer. If $\text{range} = \text{CUSOLVER_EIG_RANGE_I}$, the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq \text{il} \leq \text{iu} \leq n$, if $n &gt; 0$; $\text{il} = 1$ and $\text{iu} = 0$ if $n = 0$. Not referenced if $\text{range} = \text{CUSOLVER_EIG_RANGE_ALL} \text{ or } \text{range} = \text{CUSOLVER_EIG_RANGE_V}$.</td>
</tr>
<tr>
<td>$\text{h_meig}$</td>
<td>host</td>
<td>output</td>
<td>integer. The total number of eigenvalues found. $0 \leq \text{h_meig} \leq n$. If $\text{range} = \text{CUSOLVER_EIG_RANGE_ALL}$, $\text{h_meig} = n$, and if $\text{range} = \text{CUSOLVER_EIG_RANGE_I}$, $\text{h_meig} = \text{iu} - \text{il} + 1$.</td>
</tr>
<tr>
<td>$W$</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension $n$. The eigenvalue values of $A$, in ascending order ie, sorted so that $W(i) \leq W(i+1)$.</td>
</tr>
</tbody>
</table>
### Using the CUSOLVER API

#### 2.4.3.14. **cusolverDnSyevdx()** [DEPRECATED]

[[DEPRECATED]] use **cusolverDnXsyevdx()** instead. The routine will be removed in the next major release.

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c

cusolverStatus_t
cusolverDnSyevdx_bufferSize(
cusolverDnHandle_t handle,
cusolverParams_t params,
cusolverEigMode_t jobz,
cusolverEigRange_t range,
cublasFillMode_t uplo,
int n,
cudaDataType dataTypeA,
const void *A,
int64_t lda,
void *vl,
void *vu,
int64_t il,
int64_t iu,
int64_t *h_meig,
cudaDataType dataTypeW,
const void *W,
cudaDataType computeType,
size_t *workspaceInBytes);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>The handle to the cuSOLVER library instance.</td>
</tr>
<tr>
<td><code>params</code></td>
<td>The parameters for the calculation.</td>
</tr>
<tr>
<td><code>jobz</code></td>
<td>Specifies whether eigenvalues or eigenvectors are calculated.</td>
</tr>
<tr>
<td><code>range</code></td>
<td>Specifies the range of eigenvalues to calculate.</td>
</tr>
<tr>
<td><code>uplo</code></td>
<td>Specifies the part of the matrix A to be used.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The order of the matrix A.</td>
</tr>
<tr>
<td><code>A</code></td>
<td>The array of the matrix A.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>The leading dimension of matrix A.</td>
</tr>
<tr>
<td><code>vl</code></td>
<td>The lower vector for the eigenvectors.</td>
</tr>
<tr>
<td><code>vu</code></td>
<td>The upper vector for the eigenvectors.</td>
</tr>
<tr>
<td><code>il</code></td>
<td>The lower index for the range of eigenvalues.</td>
</tr>
<tr>
<td><code>iu</code></td>
<td>The upper index for the range of eigenvalues.</td>
</tr>
<tr>
<td><code>h_meig</code></td>
<td>The array to store the real parts of the eigenvalues.</td>
</tr>
<tr>
<td><code>computeType</code></td>
<td>The data type for the computation.</td>
</tr>
<tr>
<td><code>workspaceInBytes</code></td>
<td>The size in bytes of the workspace.</td>
</tr>
</tbody>
</table>

#### Status Returned

- **CUSOLVER_STATUS_SUCCESS**: The operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: The library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**: Invalid parameters were passed (n<0, or lda<max(1,n), or jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or range is not CUSOLVER_EIG_RANGE_ALL or CUSOLVER_EIG_RANGE_V or CUSOLVER_EIG_RANGE_I, or uplo is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER).
- **CUSOLVER_STATUS_ARCH_MISMATCH**: The device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: An internal operation failed.

---

**Note**: The CUSOLVER API is part of the cuSOLVER library, which is a part of the NVIDIA CUDA Toolkit. It provides linear algebra routines optimized for NVIDIA GPUs. This document is likely intended for developers who are familiar with CUDA and linear algebra routines.
The routine below

```c
 cusolverStatus_t
cusolverDnSyevdx ( cusolverDnHandle_t handle, cusolverParams_t params, cusolverEigMode_t jobz, cusolverEigRange_t range, cublasFillMode_t uplo, int n, cudaDataType dataTypeA, const void *A, int64_t lda, void *vl, void *vu, int64_t il, int64_t iu, int64_t *h_meig, cudaDataType dataTypeW, const void *W, cudaDataType computeType, void *pBuffer, size_t workspaceInBytes, int *info);
```

computes all or selection of the eigenvalues and optionally eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix \( \Lambda \) using the generic API interface. The standard symmetric eigenvalue problem is

\[
A \cdot V = V \cdot \Lambda
\]

where \( \Lambda \) is a real \( n \times h_{\text{meig}} \) diagonal matrix, \( V \) is an \( n \times h_{\text{meig}} \) unitary matrix, \( h_{\text{meig}} \) is the number of eigenvalues/eigenvectors computed by the routine, \( h_{\text{meig}} \) is equal to \( n \) when the whole spectrum (e.g., \( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} \)) is requested. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

The user has to provide working space which is pointed by input parameter \( \text{pBuffer} \). The input parameter \( \text{workspaceInBytes} \) is size in bytes of the working space, and it is returned by \( \text{cusolverDnSyevdx_bufferSize()} \).

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle). If \( \text{info} = i \) (greater than zero), \( i \) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

if \( \text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR} \), \( A \) contains the orthonormal eigenvectors of the matrix \( \Lambda \). The eigenvectors are computed by a divide and conquer algorithm.

Currently, \( \text{cusolverDnSyevdx} \) supports only the default algorithm.

**Table of algorithms supported by cusolverDnSyevdx**

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for \( \text{cusolverDnSyevdx_bufferSize} \) and \( \text{cusolverDnSyevdx} \):

**API of cusolverDnSyevdx**
<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>range</td>
<td>host</td>
<td>input</td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: range = CUSOLVER_EIG_RANGE_ALL: all eigenvalues/eigenvectors will be found, will becomes the classical syevd/heevd routine; range = CUSOLVER_EIG_RANGE_V: all eigenvalues/eigenvectors in the half-open interval [vl,vu] will be found; range = CUSOLVER_EIG_RANGE_I: the il-th through iu-th eigenvalues/eigenvectors will be found;</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of A is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension lda * n with lda is not less than ( \max(1,n) ). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A. On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info = 0, A contains the orthonormal eigenvectors of the matrix A. If jobz = CUSOLVER_EIG_MODE_NOVECTOR, the contents of A are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A. lda is not less than ( \max(1,n) ).</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vl, vu</td>
<td>host</td>
<td>input</td>
<td>If <code>range = CUSOLVER_EIG_RANGE_V</code>, the lower and upper bounds of the interval to be searched for eigenvalues. vl &gt; vu. Not referenced if <code>range = CUSOLVER_EIG_RANGE_ALL</code> or <code>range = CUSOLVER_EIG_RANGE_I</code>. Note that, if eigenvalues are very close to each other, it is well known that two different eigenvalues routines might find slightly different number of eigenvalues inside the same interval. This is due to the fact that different eigenvalue algorithms, or even same algorithm but different run might find eigenvalues within some rounding error close to the machine precision. Thus, if the user want to be sure not to miss any eigenvalue within the interval bound, we suggest that the user subtract/add epsilon (machine precision) to the interval bound such as <code>(vl=vl-eps, vu=vu+eps)</code>. This suggestion is valid for any selective routine from cuSolver or LAPACK.</td>
</tr>
<tr>
<td>il, iu</td>
<td>host</td>
<td>input</td>
<td>integer. If <code>range = CUSOLVER_EIG_RANGE_I</code>, the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 &lt;= il &lt;= iu &lt;= n, if n &gt; 0; il = 1 and iu = 0 if n = 0. Not referenced if <code>range = CUSOLVER_EIG_RANGE_ALL</code> or <code>range = CUSOLVER_EIG_RANGE_V</code>.</td>
</tr>
<tr>
<td>h_meig</td>
<td>host</td>
<td>output</td>
<td>integer. The total number of eigenvalues found. 0 &lt;= h_meig &lt;= n. If <code>range = CUSOLVER_EIG_RANGE_ALL</code>, h_meig = n, and if <code>range = CUSOLVER_EIG_RANGE_I</code>, h_meig = iu-il+1.</td>
</tr>
<tr>
<td>dataTypeW</td>
<td>host</td>
<td>in</td>
<td>data type of array W.</td>
</tr>
<tr>
<td>W</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension n. The eigenvalue values of A, in ascending order ie, sorted so that W(i) &lt;= W(i+1).</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>in/out</td>
<td>Working space. Array of type void of size workspaceInBytes bytes.</td>
</tr>
<tr>
<td>workspaceInBytes</td>
<td>host</td>
<td>input</td>
<td>Size in bytes of pBuffer, returned by cusolverDnSyevdx_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i (&gt; 0), info indicates i off-diagonal</td>
</tr>
</tbody>
</table>

cuSOLVER Library
The generic API has three different types, `dataTypeA` is data type of the matrix $A$, `dataTypeW` is data type of the matrix $W$ and `computeType` is compute type of the operation. `cusolverDnSyevdx` only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th><code>DataTypeA</code></th>
<th><code>DataTypeW</code></th>
<th><code>ComputeType</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SSYEVDX</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DSYEVDX</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CHEEVDX</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZHEEVDX</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ($n&lt;0$, or $lda&lt;\max(1,n)$, or $jobz$ is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODEVECTOR, or range is not CUSOLVER_EIG_RANGE_ALL or CUSOLVER_EIG_RANGE_V or CUSOLVER_EIG_RANGE_I, or $uplo$ is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.3.15. `cusolverDn<t>sygvd()`

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
void cusolverDnSygvds_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const float *A,
    int lda,
    const float *B,
    int ldb,
    const float *W,
    int *lwork);

void cusolverDnDsygvds_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const double *A,
    int lda,
    const double *B,
    int ldb,
    const double *W,
    int *lwork);
```
Using the CUSOLVER API

```c
int n,
const double *A,
int lda,
const double *B,
int ldb,
const double *W,
int *lwork);

cusolverStatus_t
cusolverDnChegvd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuComplex *A,
    int lda,
    const cuComplex *B,
    int ldb,
    const float *W,
    int *lwork);

cusolverStatus_t
cusolverDnZhegvd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *B,
    int ldb,
    const double *W,
    int *lwork);

The S and D data types are real valued single and double precision, respectively.

```c

cusolverStatus_t
cusolverDnSsygvd(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    float *B,
    int ldb,
    float *W,
    float *work,
    int lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnDsygvd(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *B,
    int ldb,
    double *W,
    double *work,
    int lwork,
```
The C and Z data types are complex valued single and double precision, respectively.

```c
int *devInfo);"
```

This function computes eigenvalues and eigenvectors of a symmetric (Hermitian) $n \times n$ matrix-pair $(A, B)$. The generalized symmetric-definite eigenvalue problem is

$$
eig(A, B) = \begin{cases} 
A*V = B*V*\Lambda & \text{if itype} = \text{CUSOLVER EIG TYPE 1} \\
A*B*V = V*\Lambda & \text{if itype} = \text{CUSOLVER EIG TYPE 2} \\
B*A*V = V*\Lambda & \text{if itype} = \text{CUSOLVER EIG TYPE 3}
\end{cases}
$$

where the matrix $B$ is positive definite, $\Lambda$ is a real $n \times n$ diagonal matrix. The diagonal elements of $\Lambda$ are the eigenvalues of $(\lambda, B)$ in ascending order. $V$ is an $n \times n$ orthogonal matrix. The eigenvectors are normalized as follows:

$$
\begin{cases} 
V^H*B*V = I & \text{if itype} = \text{CUSOLVER EIG TYPE 1, CUSOLVER EIG TYPE 2} \\
V^H*\text{inv}(B)*V = I & \text{if itype} = \text{CUSOLVER EIG TYPE 3}
\end{cases}
$$

The user has to provide working space which is pointed by input parameter `work`. The input parameter `lwork` is size of the working space, and it is returned by `sygvd_bufferSize()`.

If output parameter `devInfo` = -i (less than zero), the i-th parameter is wrong (not counting handle). If `devInfo` = i (i > 0 and i<=n) and `jobz` = CUSOLVER_EIG_MODE_NOVECTOR, i off-diagonal elements of an intermediate tridiagonal
form did not converge to zero. If devInfo = N + i (i > 0), then the leading minor of order i of 
B is not positive definite. The factorization of B could not be completed and no eigenvalues or 
eigenvectors were computed.

if jobz = CUSOLVER_EIG_MODE_VECTOR, A contains the orthogonal eigenvectors of the
matrix A. The eigenvectors are computed by divide and conquer algorithm.

Appendix F.2 provides a simple example of sygvd.

API of sygvd

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
</tbody>
</table>
| itype     | host   | input  | Specifies the problem type to be solved:

itype = CUSOLVER_EIG_TYPE_1: A*x = [lambda]*B*x.
itype = CUSOLVER_EIG_TYPE_2: A*B*x = [lambda]*x.
itype = CUSOLVER_EIG_TYPE_3: B*A*x = [lambda]*x. |
| jobz      | host   | input  | specifies options to either compute
eigenvalue only or compute eigen-pair:
jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only;
jobz = CUSOLVER_EIG_MODE_VECTOR: Compute

eigenvalues and eigenvectors. |
| uplo      | host   | input  | specifies which part of A
and B are stored. uplo =
CUBLAS_FILL_MODE_LOWER: Lower
triangle of A and B are stored.
uplo =
CUBLAS_FILL_MODE_UPPER: Upper
triangle of A and B are stored. |
| n         | host   | input  | number of rows [or columns] of matrix A
and B. |
| A         | device | in/out | <type> array of dimension lda * n
with lda is not less than max(1,n).
If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular
part of A contains the upper triangular
part of the matrix A. If uplo =
CUBLAS_FILL_MODE_LOWER, the
leading n-by-n lower triangular part
of A contains the lower triangular
part of the matrix A. On exit, if jobz
= CUSOLVER_EIG_MODE_VECTOR, and
devInfo = 0, A contains the orthonormal
eigenvectors of the matrix A. If jobz =
CUSOLVER_EIG_MODE_NOVECTOR, the
contents of A are destroyed. |
| lda       | host   | input  | leading dimension of two-dimensional
array used to store matrix A. lda is not
less than max(1,n). |
Using the CUSOLVER API

### B

- **device**: in/out
- **in**: array of dimension ldb * n.
- **out**: if uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of B contains the upper triangular part of the matrix B. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of B contains the lower triangular part of the matrix B. On exit, if devInfo is less than n, B is overwritten by triangular factor U or L from the Cholesky factorization of B.

### ldb

- **host**: input
- **in**: leading dimension of two-dimensional array used to store matrix B. ldb is not less than max(1,n).

### W

- **device**: output
- **in**: real array of dimension n. The eigenvalue values of A, sorted so that W(i) >= W(i+1).

### work

- **device**: in/out
- **in**: working space, <type> array of size lwork.

### Lwork

- **host**: input
- **in**: size of work, returned by sygvd_bufferSize.

### devInfo

- **device**: output
- **in**: if devInfo = 0, the operation is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo = i (> 0), devInfo indicates either potrf or syevd is wrong.

### Status Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.3.16. cusolverDn<t>sygvdx()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c

cusolverStatus_t
cusolverDnSygvdx_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
)```

### cuSOLVER Library

DU-06709-001_v11.1 | 168
const float *A,
int lda,
const float *B,
int ldb,
float vl,
float vu,
int il,
int iu,
int *h_meig,
const float *W,
int *lwork);

cusolverStatus_t
 cusolverDnDsygvdx_bufferSize(
   cusolverDnHandle_t handle,
   cusolverEigType_t itype,
   cusolverEigMode_t jobz,
   cusolverEigRange_t range,
   cublasFillMode_t uplo,
   int n,
   const double *A,
   int lda,
   const double *B,
   int ldb,
double vl,
double vu,
int il,
int iu,
int *h_meig,
const double *W,
int *lwork);

cusolverStatus_t
 cusolverDnChegvdx_bufferSize(
   cusolverDnHandle_t handle,
   cusolverEigType_t itype,
   cusolverEigMode_t jobz,
   cusolverEigRange_t range,
   cublasFillMode_t uplo,
   int n,
   const cuComplex *A,
   int lda,
   const cuComplex *B,
   int ldb,
float vl,
float vu,
int il,
int iu,
int *h_meig,
const float *W,
int *lwork);

cusolverStatus_t
 cusolverDnZhegvdx_bufferSize(
   cusolverDnHandle_t handle,
   cusolverEigType_t itype,
   cusolverEigMode_t jobz,
   cusolverEigRange_t range,
   cublasFillMode_t uplo,
   int n,
   const cuDoubleComplex *A,
   int lda,
   const cuDoubleComplex *B,
   int ldb,
double vl,
double vu,
int il,
int iu,
int *h_meig,
const double *W,
    int *lwork);

The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnSsygvdx(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    float *B,
    int ldb,
    float vl,
    float vu,
    int il,
    int iu,
    int *h_meig,
    float *W,
    float *work,
    int lwork,
    int *devInfo);
```

```c
cusolverStatus_t
cusolverDnDsygvdx(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *B,
    int ldb,
    double vl,
    double vu,
    int il,
    int iu,
    int *h_meig,
    double *W,
    double *work,
    int lwork,
    int *devInfo);
```

The C and Z data types are complex valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnChegvdx(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    cuComplex *B,
    int ldb,
    float vl,
    float vu,
    ...)
```
int il,
int iu,
int *h_meig,
float *W,
cuComplex *work,
int lwork,
int *devInfo);

cusolverStatus_t
cusolverDnZhegvdx(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    cuDoubleComplex *B,
    int ldb,
    double vl,
    double vu,
    int il,
    int iu,
    int *h_meig,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *devInfo);

This function computes all or selection of the eigenvalues and optionally eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix-pair \( (A, B) \). The generalized symmetric-definite eigenvalue problem is

\[
eig(A,B) = \begin{cases} 
    A^* V = B^* V \Lambda & \text{if itype = CUSOLVER_EIG_TYPE_1} \\
    A^* B^* V = V^* \Lambda & \text{if itype = CUSOLVER_EIG_TYPE_2} \\
    B^* A^* V = V^* \Lambda & \text{if itype = CUSOLVER_EIG_TYPE_3}
\end{cases}
\]

where the matrix \( B \) is positive definite. \( \Lambda \) is a real \( n \times h_{\text{meig}} \) diagonal matrix. The diagonal elements of \( \Lambda \) are the eigenvalues of \( (A, B) \) in ascending order. \( V \) is an \( n \times h_{\text{meig}} \) orthogonal matrix. \( h_{\text{meig}} \) is the number of eigenvalues/eigenvectors computed by the routine, \( h_{\text{meig}} \) is equal to \( n \) when the whole spectrum [e.g., range = CUSOLVER_EIG_RANGE_ALL] is requested. The eigenvectors are normalized as follows:

\[
\begin{cases} 
    V^H * B * V = I & \text{if itype = CUSOLVER_EIG_TYPE_1, CUSOLVER_EIG_TYPE_2} \\
    V^H * \text{inv}(B) * V = I & \text{if itype = CUSOLVER_EIG_TYPE_3}
\end{cases}
\]

The user has to provide working space which is pointed by input parameter \( \text{work} \). The input parameter \( lwork \) is size of the working space, and it is returned by \( \text{sygvdx_bufferSize()} \).

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-th parameter is wrong [not counting handle]. If \( \text{devInfo} = i \) [\( i > 0 \) and \( i \leq n \)] and \( \text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR} \), \( i \) off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If \( \text{devInfo} = n + i \) [\( i > 0 \)], then the leading minor of order \( i \) of \( B \) is not positive definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.
if jobz = CUSOLVER_EIG_MODE_VECTOR, A contains the orthogonal eigenvectors of the matrix A. The eigenvectors are computed by divide and conquer algorithm.

Appendix F.2 provides a simple example of sygvdx.

**API of sygvdx**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>itype</td>
<td>host</td>
<td>input</td>
<td>specifies the problem type to be solved: itype=CUSOLVER_EIG_TYPE_1: A<em>x = (lambda)<em>B</em>x. itype=CUSOLVER_EIG_TYPE_2: A</em>B<em>x = (lambda)<em>x. itype=CUSOLVER_EIG_TYPE_3: B</em>A</em>x = (lambda)*x.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR : Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR : Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>range</td>
<td>host</td>
<td>input</td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: range = CUSOLVER_EIG_RANGE_ALL : all eigenvalues/eigenvectors will be found, will becomes the classical syevd/heevd routine; range = CUSOLVER_EIG_RANGE_V : all eigenvalues/eigenvectors in the half-open interval [vl,vu] will be found; range = CUSOLVER_EIG_RANGE_I : the il-th through iu-th eigenvalues/eigenvectors will be found;</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of A and B are stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A and B are stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A and B are stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows [or columns] of matrix A and B.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of dimension lda * n with lda is not less than max(1,n). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of A contains the lower triangular...</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

DU-06709-001_v11.1   |   173

Using cuSOLVER

The cuSOLVER library includes the CUSOLVER API which provides a wide range of linear algebra routines optimized for NVIDIA GPUs. This includes routines for solving systems of linear equations, eigenvalue problems, singular value decompositions, etc. The CUSOLVER API enables users to leverage the power of GPUs to significantly speed up computations that are otherwise bottlenecked by CPU performance.

### Example Usage

```c
#include <cusolverDn.h>

int main()
{
    cusolverDnHandle_t handle;
    .......
   cusolverDnCreate(&handle);
    .......
    cusolverDnSetStream(handle, stream);
    .......
    cusolverDnDpotrf(handle, N, A, lda, ipiv);
    .......
    cusolverDnDestroy(handle);
    .......
    return 0;
}
```

### Parameters

#### lda
- **host**
- **input**

Leading dimension of two-dimensional array used to store matrix A. `lda` is not less than `max(1,n)`.

#### B
- **device**
- **in/out**

<? type ?> array of dimension `ldb * n`. If `uplo = CUBLAS_FILL_MODE_UPPER`, the leading `n`-by-`n` upper triangular part of `B` contains the upper triangular part of the matrix `B`. If `uplo = CUBLAS_FILL_MODE_LOWER`, the leading `n`-by-`n` lower triangular part of `B` contains the lower triangular part of the matrix `B`. On exit, if `devInfo` is less than `n`, `B` is overwritten by triangular factor `U` or `L` from the Cholesky factorization of `B`.

#### ldb
- **host**
- **input**

Leading dimension of two-dimensional array used to store matrix B. `ldb` is not less than `max(1,n)`.

#### vl,vu
- **host**
- **input**

Real values float or double for `[C, S]` or `[Z, D]` precision respectively. If `range = CUSOLVER_EIG_RANGE_V`, the lower and upper bounds of the interval to be searched for eigenvalues. `vl > vu`. Not referenced if `range = CUSOLVER_EIG_RANGE_ALL` or `range = CUSOLVER_EIG_RANGE_I`. Note that, if eigenvalues are very close to each other, it is well known that two different eigenvalues routines might find slightly different number of eigenvalues inside the same interval. This is due to the fact that different eigenvalue algorithms, or even same algorithm but different run might find eigenvalues within some rounding error close to the machine precision. Thus, if the user want to be sure not to miss any eigenvalue within the interval bound, we suggest that, the user substract/add epsilon (machine precision) to the interval bound such as `(vl=vl-eps, vu=vu+eps)`. This suggestion is valid for any selective routine from cuSolver or LAPACK.

#### il,iu
- **host**
- **input**

Integer. If `range = CUSOLVER_EIG_RANGE_I`, the indices (in ascending order) of the smallest and largest eigenvalues to be returned.
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_meig</td>
<td>integer. The total number of eigenvalues found. 0 &lt;= h_meig &lt;= n. If range = CUSOLVER_EIG_RANGE_ALL or range = CUSOLVER_EIG_RANGE_V, h_meig = n, and if range = CUSOLVER_EIG_RANGE_I, h_meig = iu-il+1.</td>
</tr>
<tr>
<td>W</td>
<td>a real array of dimension n. The eigenvalue values of A, sorted so that W(i) &gt;= W(i+1).</td>
</tr>
<tr>
<td>work</td>
<td>working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>size of work, returned by sygvdx_bufferSize.</td>
</tr>
<tr>
<td>devInfo</td>
<td>if devInfo = 0, the operation is successful. if devInfo = -i, the i-th parameter is wrong [not counting handle]. if devInfo = i (&gt; 0), devInfo indicates either potrf or syevd is wrong.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed (n<0, or lda<max(1,n), or ldb<max(1,n), or itype is not CUSOLVER_EIG_TYPE_1 or CUSOLVER_EIG_TYPE_2 or CUSOLVER_EIG_TYPE_3 or jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTORL, or range is not CUSOLVER_EIG_RANGE_ALL or CUSOLVER_EIG_RANGE_V or CUSOLVER_EIG_RANGE_I, or uplo is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER). |
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.

### 2.4.3.17. cusolverDn<t>syevj() 

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
 cusolverDnSsyevj_bufferSize( 
     cusolverDnHandle_t handle, 
     cusolverEigMode_t jobz, 
     cublasFillMode_t uplo, 
```
int n,
const float *A,
int lda,
const float *W,
int *lwork,
syevjInfo_t params);

cusolverStatus_t
cusolverDnDsyevj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const double *A,
    int lda,
    const double *W,
    int *lwork,
    syevjInfo_t params);

cusolverStatus_t
cusolverDnCheevj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuComplex *A,
    int lda,
    const float *W,
    int *lwork,
    syevjInfo_t params);

cusolverStatus_t
cusolverDnZheevj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const cuDoubleComplex *A,
    int lda,
    const double *W,
    int *lwork,
    syevjInfo_t params);

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
 cusolverDnSsyevj(
     cusolverDnHandle_t handle,
     cusolverEigMode_t jobz,
     cublasFillMode_t uplo,
     int n,
     float *A,
     int lda,
     float *W,
     float *work,
     int lwork,
     int *info,
     syevjInfo_t params);

cusolverStatus_t
 cusolverDnDsyevj(
     cusolverDnHandle_t handle,
     cusolverEigMode_t jobz,
     cublasFillMode_t uplo,
     int n,
     double *A,
     int lda,
     double *W,
     double *work,
Using the CUSOLVER API

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnCheevj(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    float *W,
    cuComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);

cusolverStatus_t
cusolverDnZheevj(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);

This function computes eigenvalues and eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix \( A \).

The standard symmetric eigenvalue problem is

\[
A Q = Q \Lambda
\]

where \( \Lambda \) is a real \( n \times n \) diagonal matrix, \( Q \) is an \( n \times n \) unitary matrix. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

\( \text{syevj} \) has the same functionality as \( \text{syevd} \). The difference is that \( \text{syevd} \) uses QR algorithm and \( \text{syevj} \) uses Jacobi method. The parallelism of Jacobi method gives GPU better performance on small and medium size matrices. Moreover the user can configure \( \text{syevj} \) to perform approximation up to certain accuracy.

How does it work?

\( \text{syevj} \) iteratively generates a sequence of unitary matrices to transform matrix \( A \) to the following form

\[
V^H A V = W + E
\]

where \( W \) is diagonal and \( E \) is symmetric without diagonal.

During the iterations, the Frobenius norm of \( E \) decreases monotonically. As \( E \) goes down to zero, \( W \) is the set of eigenvalues. In practice, Jacobi method stops if
where $\text{eps}$ is given tolerance.

`syevj` has two parameters to control the accuracy. First parameter is tolerance ($\text{eps}$). The default value is machine accuracy but the user can use function `cusolverDnXsyevjSetTolerance` to set a priori tolerance. The second parameter is maximum number of sweeps which controls number of iterations of Jacobi method. The default value is 100 but the user can use function `cusolverDnXsyevjSetMaxSweeps` to set a proper bound. The experiments show 15 sweeps are good enough to converge to machine accuracy. `syevj` stops either tolerance is met or maximum number of sweeps is met.

Jacobi method has quadratic convergence, so the accuracy is not proportional to number of sweeps. To guarantee certain accuracy, the user should configure tolerance only.

After `syevj`, the user can query residual by function `cusolverDnXsyevjGetResidual` and number of executed sweeps by function `cusolverDnXsyevjGetSweeps`. However the user needs to be aware that residual is the Frobenius norm of $E$, not accuracy of individual eigenvalue, i.e.

$$\text{residual} = ||E||_F = ||N - W||_F$$

The same as `syevd`, the user has to provide working space pointed by input parameter `work`. The input parameter `lwork` is the size of the working space, and it is returned by `syevj_bufferSize()`.

If output parameter `info = -i` (less than zero), the $i$-th parameter is wrong (not counting handle). If `info = n+1`, `syevj` does not converge under given tolerance and maximum sweeps.

If the user sets an improper tolerance, `syevj` may not converge. For example, tolerance should not be smaller than machine accuracy.

If `jobz = CUSOLVER_EIG_MODE_VECTOR`, $A$ contains the orthonormal eigenvectors $V$.

Appendix F.3 provides a simple example of `syevj`.

**API of syevj**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td><code>jobz</code></td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: $\text{jobz} = CUSOLVER_EIG_MODE_NOVECTOR: \text{Compute eigenvalues only}$; $\text{jobz} = CUSOLVER_EIG_MODE_VECTOR: \text{Compute eigenvalues and eigenvectors.}$</td>
</tr>
<tr>
<td><code>uplo</code></td>
<td>host</td>
<td>input</td>
<td>specifies which part of $A$ is stored. $\text{uplo} = CUBLAS_FILL_MODE_LOWER: \text{Lower triangle of } A \text{ is stored. }$ $\text{uplo} = CUBLAS_FILL_MODE_UPPER: \text{Upper triangle of } A \text{ is stored.}$</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>number of rows [or columns] of matrix $A$.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

A | device | in/out | <type> array of dimension lda * n with lda is not less than max(1,n).
   |        |        | If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A. On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info = 0, A contains the orthonormal eigenvectors of the matrix A. If jobz = CUSOLVER_EIG_MODE_NOVECTOR, the contents of A are destroyed.

lda | host | input | leading dimension of two-dimensional array used to store matrix A.

W | device | output | a real array of dimension n. The eigenvalue values of A, in ascending order ie, sorted so that W(i) <= W(i+1).

work | device | in/out | working space, <type> array of size lwork.

lwork | host | input | size of work, returned by syevj_bufferSize.

info | device | output | if info = 0, the operation is successful.
   |        |        | if info = -i, the i-th parameter is wrong (not counting handle). if info = n +1, syevj dose not converge under given tolerance and maximum sweeps.

params | host | in/out | structure filled with parameters of Jacobi algorithm and results of syevj.

### Status Returned

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed \( |n < 0, \\
   or lda < max(1,n) \), or jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or uplo is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.

#### 2.4.3.18. cusolverDn<t>sygvj()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
cusolverDn&lt;sygvj_bufferSize(  ```
The S and D data types are real valued single and double precision, respectively.
Using the CUSOLVER API

```c
int lda,
float *B,
int ldb,
float *W,
float *work,
int lwork,
int *info,
syevjInfo_t params);

cusolverStatus_t
 cusolverDnDsygvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *B,
    int ldb,
    double *W,
    double *work,
    int lwork,
    int *info,
    syevjInfo_t params);

cusolverStatus_t
 cusolverDnChegvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    cuComplex *B,
    int ldb,
    float *W,
    cuComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);

cusolverStatus_t
 cusolverDnZhegvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    cuDoubleComplex *B,
    int ldb,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);
```

The C and Z data types are complex valued single and double precision, respectively.

```c

cusolverStatus_t
 cusolverDnDsygvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    double *B,
    int ldb,
    double *W,
    double *work,
    int lwork,
    int *info,
    syevjInfo_t params);

cusolverStatus_t
 cusolverDnChegvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    cuComplex *B,
    int ldb,
    float *W,
    cuComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);

cusolverStatus_t
 cusolverDnZhegvj(
    cusolverDnHandle_t handle,
    cusolverEigType_t itype,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    cuDoubleComplex *B,
    int ldb,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params);
```
This function computes eigenvalues and eigenvectors of a symmetric (Hermitian) $n \times n$ matrix-pair $\{A, B\}$. The generalized symmetric-definite eigenvalue problem is

$$
eig(A, B) = \begin{cases} 
A \times V = B \times V \times \Lambda & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE}_1 \\
A \times B \times V = V \times \Lambda & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE}_2 \\
B \times A \times V = V \times \Lambda & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE}_3
\end{cases}
$$

where the matrix $B$ is positive definite. $\Lambda$ is a real $n \times n$ diagonal matrix. The diagonal elements of $\Lambda$ are the eigenvalues of $\{A, B\}$ in ascending order. $V$ is an $n \times n$ orthogonal matrix. The eigenvectors are normalized as follows:

$$
\begin{cases}
V^H \times B \times V = I & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE}_1, \text{CUSOLVER_EIG_TYPE}_2 \\
V^H \times \text{inv}(B) \times V = I & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE}_3
\end{cases}
$$

This function has the same functionality as sygvd except that syevd in sygvd is replaced by syevj in sygvj. Therefore, sygvj inherits properties of syevj, the user can use cusolverDnXsyevjSetTolerance and cusolverDnXsyevjSetMaxSweeps to configure tolerance and maximum sweeps.

However the meaning of residual is different from syevj. sygvj first computes Cholesky factorization of matrix $B$,

$$
B = L \times L^H
$$

transform the problem to standard eigenvalue problem, then calls syevj.

For example, the standard eigenvalue problem of type I is

$$
M \times Q = Q \times \Lambda
$$

where matrix $M$ is symmetric

$$
M = L^{-1} \times A \times L^{-H}
$$

The residual is the result of syevj on matrix $M$, not $A$.

The user has to provide working space which is pointed by input parameter work. The input parameter lwork is the size of the working space, and it is returned by sygvj_bufferSize().

If output parameter info = -i (less than zero), the i-th parameter is wrong [not counting handle]. If info = i [i > 0 and i<=$n$], $B$ is not positive definite, the factorization of $B$ could not be completed and no eigenvalues or eigenvectors were computed. If info = $n$+$i$, syevj does not converge under given tolerance and maximum sweeps. In this case, the eigenvalues and eigenvectors are still computed because non-convergence comes from improper tolerance of maximum sweeps.

if jobz = CUSOLVER_EIG_MODE_VECTOR, $A$ contains the orthogonal eigenvectors $V$.

Appendix F.4 provides a simple example of sygvj.

**API of sygvj**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

| **itype** | **host** | **input** | Specifies the problem type to be solved:  
  * itype=CUSOLVER_EIG_TYPE_1: $A\times x = (\lambda)B\times x$.  
  * itype=CUSOLVER_EIG_TYPE_2: $A\times B\times x = (\lambda)\times x$.  
  * itype=CUSOLVER_EIG_TYPE_3: $B\times A\times x = (\lambda)\times x$. |
| --- | --- | --- | --- |
| **jobz** | **host** | **input** | Specifies options to either compute  
  eigenvalue only or compute eigen-pair:  
  * jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only;  
  * jobz = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors. |
| **uplo** | **host** | **input** | Specifies which part of $A$  
 and $B$ are stored.  
  * uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of $A$ and $B$ are stored.  
  * uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of $A$ and $B$ are stored. |
| **n** | **host** | **input** | Number of rows (or columns) of matrix $A$ and $B$. |
| **A** | **device** | **in/out** | \(<\text{type}>\) array of dimension $lda \times n$  
  with $lda$ is not less than $\max(1,n)$.  
  If $uplo = \text{CUBLAS_FILL_MODE_UPPER}$,  
  the leading $n$-by-$n$ upper triangular part of $A$ contains the upper triangular part of the matrix $A$.  
  If $uplo = \text{CUBLAS_FILL_MODE_LOWER}$,  
  the leading $n$-by-$n$ lower triangular part of $A$ contains the lower triangular part of the matrix $A$.  
  On exit, if $jobz = \text{CUSOLVER_EIG_MODE_VECTOR}$, and $info = 0$, $A$ contains the orthonormal eigenvectors of the matrix $A$.  
  If $jobz = \text{CUSOLVER_EIG_MODE_NOVECTOR}$, the contents of $A$ are destroyed. |
| **lda** | **host** | **input** | Leading dimension of two-dimensional array used to store matrix $A$.  
  $lda$ is not less than $\max(1,n)$. |
| **B** | **device** | **in/out** | \(<\text{type}>\) array of dimension $ldb \times n$.  
  If $uplo = \text{CUBLAS_FILL_MODE_UPPER}$,  
  the leading $n$-by-$n$ upper triangular part of $B$ contains the upper triangular part of the matrix $B$.  
  If $uplo = \text{CUBLAS_FILL_MODE_LOWER}$, the leading $n$-by-$n$ lower triangular part of $B$ contains the lower triangular part of the matrix $B$.  
  On exit, if $info$ is less than $n$, $B$ is overwritten by triangular factor $U$ or $L$  
  from the Cholesky factorization of $B$. |
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>ldb</th>
<th>host</th>
<th>input</th>
<th>leading dimension of two-dimensional array used to store matrix B. ldb is not less than max(1,n).</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension n. The eigenvalue values of A, sorted so that W(i) &gt;= W(i+1).</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of work, returned by sygvj_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i (&gt; 0), info indicates either B is not positive definite or syevj (called by sygvj) does not converge.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.3.19. cusolverDn<t>syevjBatched()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
 cusolverDnSsyevjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const float *A,
    int lda,
    const float *W,
    int *lwork,
    syevjInfo_t params,
    int batchSize
);

cusolverStatus_t
 cusolverDnDsyevjBatched_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    const double *A,
```
using the Cusolver API

```c
int lda,
const double *W,
int *lwork,
syevjInfo_t params,
int batchSize
);

cusolverStatus_t
cusolverDnCheevjBatched_bufferSize(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
const cuComplex *A,
int lda,
const float *W,
int *lwork,
syevjInfo_t params,
int batchSize
);

cusolverStatus_t
cusolverDnZheevjBatched_bufferSize(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
const cuDoubleComplex *A,
int lda,
const double *W,
int *lwork,
syevjInfo_t params,
int batchSize
);

cusolverStatus_t
cusolverDnSsyevjBatched(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
float *A,
int lda,
float *W,
float *work,
int lwork,
int *info,
syevjInfo_t params,
int batchSize
);

cusolverStatus_t
cusolverDnDsyevjBatched(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
int n,
double *A,
int lda,
double *W,
double *work,
int lwork,
int *info,
syevjInfo_t params,
int batchSize
);
```

The S and D data types are real valued single and double precision, respectively.
The C and Z data types are complex valued single and double precision, respectively.

```c
 cusolverStatus_t
cusolverDnCheevjBatched(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    float *W,
    cuComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params,
    int batchSize
);

cusolverStatus_t
cusolverDnZheevjBatched(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    double *W,
    cuDoubleComplex *work,
    int lwork,
    int *info,
    syevjInfo_t params,
    int batchSize
);
```

This function computes eigenvalues and eigenvectors of a sequence of symmetric (Hermitian)
\(n \times n\) matrices

\[ A_j^* Q_j = Q_j^* \Lambda_j \]

where \(\Lambda_j\) is a real \(n \times n\) diagonal matrix, \(Q_j\) is an \(n \times n\) unitary matrix. The diagonal elements of \(\Lambda_j\) are the eigenvalues of \(A_j\) in either ascending order or non-sorting order.

\(\text{syevjBatched}\) performs \(\text{syevj}\) on each matrix. It requires that all matrices are of the same size \(n\) and are packed in contiguous way,

\[ A = (A0 \quad A1 \quad ...) \]

Each matrix is column-major with leading dimension \(lda\), so the formula for random access is

\[ A_k(i,j) = A[i + lda*j + lda*n*k] \]

The parameter \(W\) also contains eigenvalues of each matrix in contiguous way,

\[ W = (W0 \quad W1 \quad ...) \]

The formula for random access of \(W\) is

\[ W_k(j) = W[j + n*k] \]
Except for tolerance and maximum sweeps, `syevjBatched` can either sort the eigenvalues in ascending order [default] or chose as-is [without sorting] by the function `cusolverDnXsyevjSetSortEig`. If the user packs several tiny matrices into diagonal blocks of one matrix, non-sorting option can separate spectrum of those tiny matrices.

`syevjBatched` cannot report residual and executed sweeps by function `cusolverDnXsyevjGetResidual` and `cusolverDnXsyevjGetSweeps`. Any call of the above two returns `CUSOLVER_STATUS_NOT_SUPPORTED`. The user needs to compute residual explicitly.

The user has to provide working space pointed by input parameter `work`. The input parameter `lwork` is the size of the working space, and it is returned by `syevjBatched_bufferSize()`.

The output parameter `info` is an integer array of size `batchSize`. If the function returns `CUSOLVER_STATUS_INVALID_VALUE`, the first element `info[0] = -i` (less than zero) indicates `i`-th parameter is wrong (not counting handle). Otherwise, if `info[i] = n+1`, `syevjBatched` does not converge on `i`-th matrix under given tolerance and maximum sweeps.

if `jobz = CUSOLVER_EIG_MODE_VECTOR`, $A_j$ contains the orthonormal eigenvectors $V_j$.

Appendix F.5 provides a simple example of `syevjBatched`.

### API of `syevjBatched`

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR ; Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR ; Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of $A_j$ is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of $A_j$ is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of $A_j$ is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows [or columns] of matrix each $A_j$.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>$&lt;$type$&gt;$ array of dimension <code>lda</code> * <code>n</code> * <code>batchSize</code> with <code>lda</code> is not less than max(1, <code>n</code>). If <code>uplo</code> = CUBLAS_FILL_MODE_UPPER, the leading <code>n</code>-by-<code>n</code> upper triangular part of $A_j$ contains the upper triangular part of the matrix $A_j$. If <code>uplo</code> = CUBLAS_FILL_MODE_LOWER, the leading <code>n</code>-by-<code>n</code> lower triangular part of $A_j$ contains the lower triangular part of the matrix $A_j$. On exit, if <code>jobz</code> = CUSOLVER_EIG_MODE_VECTOR, and <code>info[j] = 0</code>, $A_j$ contains</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

The orthonormal eigenvectors of the matrix \( A_j \). If \( \text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR} \), the contents of \( A_j \) are destroyed.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lda} )</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A_j ).</td>
</tr>
<tr>
<td>( \text{W} )</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension ( n \times \text{batchSize} ). It stores the eigenvalues of ( A_j ) in ascending or non-sorting order.</td>
</tr>
<tr>
<td>( \text{work} )</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) array of size ( \text{lwork} ), workspace.</td>
</tr>
<tr>
<td>( \text{lwork} )</td>
<td>host</td>
<td>input</td>
<td>size of ( \text{work} ), returned by ( \text{syevjBatched_bufferSize} ).</td>
</tr>
<tr>
<td>( \text{info} )</td>
<td>device</td>
<td>output</td>
<td>an integer array of dimension ( \text{batchSize} ). If ( \text{CUSOLVER_STATUS_INVALID_VALUE} ) is returned, ( \text{info}[0] = -i ) (less than zero) indicates ( i )-th parameter is wrong [not counting handle]. Otherwise, if ( \text{info}[i] = 0 ), the operation is successful. If ( \text{info}[i] = n+1 ), ( \text{syevjBatched} ) does not converge on ( i )-th matrix under given tolerance and maximum sweeps.</td>
</tr>
<tr>
<td>( \text{params} )</td>
<td>host</td>
<td>in/out</td>
<td>structure filled with parameters of Jacobi algorithm.</td>
</tr>
<tr>
<td>( \text{batchSize} )</td>
<td>host</td>
<td>input</td>
<td>number of matrices.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (( n &lt; 0 ), ( \text{lda} &lt; \max(1, n) ), ( \text{jobz} ) is not ( \text{CUSOLVER_EIG_MODE_NOVECTOR} ) or ( \text{CUSOLVER_EIG_MODE_VECTOR} ), or ( \text{uplo} ) is not ( \text{CUBLAS_FILL_MODE_LOWER} ) or ( \text{CUBLAS_FILL_MODE_UPPER} ), or ( \text{batchSize} &lt; 0 )).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.4. Dense Linear Solver Reference (64-bit API)

This chapter describes linear solver 64-bit API of cuSolverDN, including Cholesky factorization, LU with partial pivoting and QR factorization.

#### 2.4.4.1. cusolverDnXpotrf()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c

// Define cusolverDnXpotrf_bufferSize function

```
The routine below

```c
 cusolverStatus_t
cusolverDnXpotrf(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasFillMode_t uplo,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType computeType,
    void *bufferOnDevice,
    size_t workspaceInBytesOnDevice,
    void *bufferOnHost,
    size_t workspaceInBytesOnHost,
    int *info )
```

computes the Cholesky factorization of a Hermitian positive-definite matrix using the generic API interface.

A is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful. The input parameter `uplo` indicates which part of the matrix is used. The function would leave other part untouched.

If input parameter `uplo` is `CUBLAS_FILL_MODE_LOWER`, only lower triangular part of \( A \) is processed, and replaced by lower triangular Cholesky factor \( L \).

\[
A = L \cdot L^H
\]

If input parameter `uplo` is `CUBLAS_FILL_MODE_UPPER`, only upper triangular part of \( A \) is processed, and replaced by upper triangular Cholesky factor \( U \).

\[
A = U^H \cdot U
\]

The user has to provide device and host working spaces which are pointed by input parameters `bufferOnDevice` and `bufferOnHost`. The input parameters `workspaceInBytesOnDevice` (and `workspaceInBytesOnHost`) is size in bytes of the device (and host) working space, and it is returned by `cusolverDnXpotrf_bufferSize()`.

If Cholesky factorization failed, i.e. some leading minor of \( A \) is not positive definite, or equivalently some diagonal elements of \( L \) or \( U \) is not a real number. The output parameter `info` would indicate smallest leading minor of \( A \) which is not positive definite.

If output parameter `info` = -\( i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

Currently, `cusolverDnXpotrf` supports only the default algorithm.
Using the CUSOLVER API

Table of algorithms supported by cusolverDnXpotrf

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for cusolverDnXpotrf_bufferSize and cusolverDnXpotrf:

API of potrf

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix A lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension lda * n with lda is not less than max(1, n).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>bufferOnDevice</td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type void of size workspaceInBytesOnDevice bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnDevice</td>
<td>host</td>
<td>input</td>
<td>size in bytes of bufferOnDevice, returned by cusolverDnXpotrf_bufferSize.</td>
</tr>
<tr>
<td>bufferOnHost</td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type void of size workspaceInBytesOnHost bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnHost</td>
<td>host</td>
<td>input</td>
<td>size in bytes of bufferOnHost, returned by cusolverDnXpotrf_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the Cholesky factorization is successful. if info = -i, the i-th parameter is wrong [not counting handle]. if info = i, the leading minor of order i is not positive definite.</td>
</tr>
</tbody>
</table>

The generic API has two different types, dataTypeA is data type of the matrix A, computeType is compute type of the operation. cusolverDnXpotrf only supports the following four combinations.

Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRF</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRF</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRF</td>
</tr>
</tbody>
</table>
Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( n&lt;0 ) or ( lda&lt;\max(1,n) ).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.4.2. cusolverDnXpotrs()

cusolverStatus_t
cusolverDnXpotrs(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasFillMode_t uplo,
    int64_t n,
    int64_t nrhs,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeB,
    void *B,
    int64_t ldb,
    int *info)

This function solves a system of linear equations

\[
A^* X = B
\]

where \( A \) is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful using the generic API interface. The input parameter \( \text{uplo} \) indicates which part of the matrix is used. The function would leave other part untouched.

The user has to call \( \text{cusolverDnXpotrf} \) first to factorize matrix \( A \). If input parameter \( \text{uplo} \) is CUBLAS_FILL_MODE_LOWER, \( A \) is lower triangular Cholesky factor \( L \) corresponding to \( A = L^* L^H \). If input parameter \( \text{uplo} \) is CUBLAS_FILL_MODE_UPPER, \( A \) is upper triangular Cholesky factor \( U \) corresponding to \( A = U^H U^* \).

The operation is in-place, i.e. matrix \( X \) overwrites matrix \( B \) with the same leading dimension \( \text{ldb} \).

If output parameter \( \text{info} = -i \) (less than zero), the \( i-th \) parameter is wrong (not counting handle).

Currently, \( \text{cusolverDnXpotrs} \) supports only the default algorithm.

Table of algorithms supported by \( \text{cusolverDnXpotrs} \)

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

List of input arguments for \( \text{cusolverDnXpotrs} \):

API of potrs

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolveDN library context.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>params</th>
<th>host</th>
<th>input</th>
<th>structure with information collected by cusolverDnSetAdvOptions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix A lower or upper part is stored, the other part is not referenced.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix X and B.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>input</td>
<td>array of dimension lda * n with lda is not less than max(1,n). A is either lower cholesky factor L or upper Cholesky factor U.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>dataTypeB</td>
<td>host</td>
<td>in</td>
<td>data type of array B.</td>
</tr>
<tr>
<td>B</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension ldb * nrhs. ldb is not less than max(1,n). As an input, B is right hand side matrix. As an output, B is the solution matrix.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the Cholesky factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, dataTypeA is data type of the matrix A, dataTypeB is data type of the matrix B. cusolverDnXpotrs only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>dataTypeA</th>
<th>dataTypeB</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRS</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRS</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRS</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRS</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [n&lt;0, nrhs&lt;0, lda&lt;max(1,n) or ldb&lt;max(1,n)].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

**2.4.4.3. cusolverDnXgetrf()**

The helper function below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t cusolverDnXgetrf_bufferSize(
```
The function below computes the LU factorization of a \( m \times n \) matrix

\[
P \cdot A = L \cdot U
\]

where \( A \) is a \( m \times n \) matrix, \( P \) is a permutation matrix, \( L \) is a lower triangular matrix with unit diagonal, and \( U \) is an upper triangular matrix using the generic API interface.

If LU factorization failed, i.e. matrix \( A \) (or \( U \)) is singular, the output parameter \( \text{info}=i \) indicates \( U(i,i) = 0 \).

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

If \( \text{ipiv} \) is null, no pivoting is performed. The factorization is \( A=L\cdot U \), which is not numerically stable.

No matter LU factorization failed or not, the output parameter \( \text{ipiv} \) contains pivoting sequence, row \( i \) is interchanged with row \( \text{ipiv}(i) \).

The user has to provide device and host working spaces which are pointed by input parameters \( \text{bufferOnDevice} \) and \( \text{bufferOnHost} \). The input parameters \( \text{workspaceInBytesOnDevice} \) (and \( \text{workspaceInBytesOnHost} \)) is size in bytes of the device (and host) working space, and it is returned by \( \text{cusolverDnXgetrf_bufferSize()} \).

The user can combine \( \text{cusolverDnXgetrf} \) and \( \text{cusolverDnGetrs} \) to complete a linear solver.

Currently, \( \text{cusolverDnXgetrf} \) supports two algorithms. To select legacy implementation, the user has to call \( \text{cusolverDnSetAdvOptions} \).

**Table of algorithms supported by cusolverDnXgetrf**
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>CUSOLVER_ALG_0 or NULL</th>
<th>Default algorithm. The fastest, requires a large workspace of ( m \times n ) elements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_ALG_1</td>
<td>Legacy implementation</td>
</tr>
</tbody>
</table>

List of input arguments for `cusolverDnXgetrf_bufferSize` and `cusolverDnXgetrf`:

**API of cusolverDnXgetrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by <code>cusolverDnSetAdvOptions</code>.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array ( A ).</td>
</tr>
<tr>
<td>( A )</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) array of dimension ( lda \times n ) with ( lda ) is not less than ( \max(1,m) ).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td>ipiv</td>
<td>device</td>
<td>output</td>
<td>array of size at least ( \min(m,n) ), containing pivot indices.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>bufferOnDevice</td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type void of size <code>workspaceInBytesOnDevice</code> bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnDevice</td>
<td>input</td>
<td></td>
<td>size in bytes of <code>bufferOnDevice</code>, returned by <code>cusolverDnXpotrf_bufferSize</code>.</td>
</tr>
<tr>
<td>bufferOnHost</td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type void of size <code>workspaceInBytesOnHost</code> bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnHost</td>
<td>input</td>
<td></td>
<td>size in bytes of <code>bufferOnHost</code>, returned by <code>cusolverDnXpotrf_bufferSize</code>.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if ( info = 0 ), the LU factorization is successful. if ( info = -i ), the ( i )-th parameter is wrong [not counting handle]. if ( info = i ), the ( U(i,i) = 0 ).</td>
</tr>
</tbody>
</table>

The generic API has two different types, `dataTypeA` is data type of the matrix \( A \), `computeType` is compute type of the operation. `cusolverDnXgetrf` only supports the following four combinations.

**valid combination of data type and compute type**

<table>
<thead>
<tr>
<th><code>DataTypeA</code></th>
<th><code>ComputeType</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGETRF</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGETRF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGETRF</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGETRF</td>
</tr>
</tbody>
</table>
Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.4.4. cusolverDnXgetrs()

cusolverStatus_t

```c

cusolverDnXgetrs(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cublasOperation_t trans,
    int64_t n,
    int64_t nrhs,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    const int64_t *ipiv,
    cudaDataType dataTypeB,
    void *B,
    int64_t ldb,
    int *info )
```

This function solves a linear system of multiple right-hand sides

\[ \text{op}(\mathbf{A}) \mathbf{X} = \mathbf{B} \]

where \(\mathbf{A}\) is a \(n \times n\) matrix, and was LU-factored by \text{cusolverDnXgetrf}, that is, lower triangular part of \(\mathbf{A}\) is \(\mathbf{L}\), and upper triangular part (including diagonal elements) of \(\mathbf{A}\) is \(\mathbf{U}\). \(\mathbf{B}\) is a \(n \times nrhs\) right-hand side matrix using the generic API interface.

The input parameter \text{trans} is defined by

\[
\text{op}(\mathbf{A}) = \begin{cases} 
\mathbf{A} & \text{if trans} = \text{CUBLAS.OP.N} \\
\mathbf{A}^T & \text{if trans} = \text{CUBLAS.OP.T} \\
\mathbf{A}^H & \text{if trans} = \text{CUBLAS.OP.C} 
\end{cases}
\]

The input parameter \text{ipiv} is an output of \text{cusolverDnXgetrf}. It contains pivot indices, which are used to permutate right-hand sides.

If output parameter \text{info} = -i (less than zero), the \(i\)-th parameter is wrong (not counting handle).

The user can combine \text{cusolverDnXgetrf} and \text{cusolverDnXgetrs} to complete a linear solver.

Currently, \text{cusolverDnXgetrs} supports only the default algorithm.

**Table of algorithms supported by cusolverDnXgetrs**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_ALG_0 or NULL</td>
<td>Default algorithm.</td>
</tr>
</tbody>
</table>

List of input arguments for \text{cusolverDnXgetrs}:
<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>trans</td>
<td>host</td>
<td>input</td>
<td>operation op(A) that is non- or (conj.) transpose.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>number of right-hand sides.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>input</td>
<td>array of dimension lda * n with lda is not less than max(1, n).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>ipiv</td>
<td>device</td>
<td>input</td>
<td>array of size at least n, containing pivot indices.</td>
</tr>
<tr>
<td>dataTypeB</td>
<td>host</td>
<td>in</td>
<td>data type of array B.</td>
</tr>
<tr>
<td>B</td>
<td>device</td>
<td>output</td>
<td>array of dimension ldb * nrhs with ldb is not less than max(1, n).</td>
</tr>
<tr>
<td>ldb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix B.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, dataTypeA is data type of the matrix A and dataTypeB is data type of the matrix B. cusolverDnXgetrs only supports the following four combinations.

### Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>dataTypeB</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGETRS</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGETRS</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGETRS</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGETRS</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [n&lt;0 or lda&lt;max(1, n) or ldb&lt;max(1, n)].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
2.4.4.5. cusolverDnXgeqrf()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverDnXgeqrf_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeTau,
    const void *tau,
    cudaDataType computeType,
    size_t *workspaceInBytesOnDevice,
    size_t *workspaceInBytesOnHost)
```

The routine below

```c
cusolverStatus_t cusolverDnXgeqrf(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType dataTypeTau,
    void *tau,
    cudaDataType computeType,
    void *bufferOnDevice,
    size_t workspaceInBytesOnDevice,
    void *bufferOnHost,
    size_t workspaceInBytesOnHost,
    int *info )
```

computes the QR factorization of a $m \times n$ matrix

$$A = Q R$$

where $A$ is a $m \times n$ matrix, $Q$ is a $m \times n$ matrix, and $R$ is a $n \times n$ upper triangular matrix using the generic API interface.

The user has to provide device and host working spaces which are pointed by input parameters `bufferOnDevice` and `bufferOnHost`. The input parameters `workspaceInBytesOnDevice` (and `workspaceInBytesOnHost`) is size in bytes of the device (and host) working space, and it is returned by `cusolverDnXgeqrf_bufferSize()`.

The matrix $R$ is overwritten in upper triangular part of $A$, including diagonal elements.

The matrix $Q$ is not formed explicitly, instead, a sequence of householder vectors are stored in lower triangular part of $A$. The leading nonzero element of householder vector is assumed to be 1 such that output parameter $TAU$ contains the scaling factor $\tau$. If $v$ is original householder vector, $q$ is the new householder vector corresponding to $\tau$, satisfying the following relation

$$I - 2vv^H = I - \tau qq^H$$
If output parameter info = -i (less than zero), the i-th parameter is wrong (not counting handle).

Currently, cusolverDnXgeqrf supports only the default algorithm.

**Table of algorithms supported by cusolverDnXgeqrf**

<table>
<thead>
<tr>
<th>CUSOLVER_ALG_0 or NULL</th>
<th>Default algorithm.</th>
</tr>
</thead>
</table>

**List of input arguments for cusolverDnXgeqrf_bufferSize and cusolverDnXgeqrf:**

**API of geqrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension lda * n with lda is not less than max(1,m).</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix A.</td>
</tr>
<tr>
<td>TAU</td>
<td>device</td>
<td>output</td>
<td>array of dimension at least min(m,n).</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>bufferOnDevice</td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type void of size workspaceInBytesOnDevice bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnDevice</td>
<td>input</td>
<td></td>
<td>size in bytes of bufferOnDevice, returned by cusolverDnXpotrf_bufferSize.</td>
</tr>
<tr>
<td>bufferOnHost</td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type void of size workspaceInBytesOnHost bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnHost</td>
<td>input</td>
<td></td>
<td>size in bytes of bufferOnHost, returned by cusolverDnXpotrf_bufferSize.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if info = 0, the LU factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

The generic API has two different types, dataTypeA is data type of the matrix A and array tau and computeType is compute type of the operation. cusolverDnXgeqrf only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGEQRF</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGEQRF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGEQRF</td>
</tr>
</tbody>
</table>
2.4.5. Dense Eigenvalue Solver Reference (64-bit API)

This chapter describes eigenvalue solver API of cuSolverDN, including bidiagonalization and SVD.

2.4.5.1. cusolverDnXgesvd()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
 cusolverStatus_t
 cusolverDnXgesvd_bufferSize(
     cusolverDnHandle_t handle,
     cusolverDnParams_t params,
     signed char jobu,
     signed char jobvt,
     int64_t m,
     int64_t n,
     cudaDataType dataTypeA,
     const void *A,
     int64_t lda,
     cudaDataType dataTypeS,
     const void *S,
     cudaDataType dataTypeU,
     const void *U,
     int64_t ldu,
     cudaDataType dataTypeVT,
     const void *VT,
     int64_t ldvt,
     cudaDataType computeType,
     size_t *workspaceInBytesOnDevice,
     size_t *workspaceInBytesOnHost)
```

The routine below

```c
 cusolverStatus_t
 cusolverDnXgesvd(
     cusolverDnHandle_t handle,
     cusolverDnParams_t params,
     signed char jobu,
     signed char jobvt,
     int64_t m,
     int64_t n,
     cudaDataType dataTypeA,
     void *A,
     int64_t lda,
     cudaDataType dataTypeS,
     void *S,
     cudaDataType dataTypeU,
```
This function computes the singular value decomposition (SVD) of a $m \times n$ matrix $A$ and corresponding the left and/or right singular vectors. The SVD is written

$$ A = U \Sigma V^H $$

where $\Sigma$ is an $m \times n$ matrix which is zero except for its $\min(m, n)$ diagonal elements, $U$ is an $m \times m$ unitary matrix, and $V$ is an $n \times n$ unitary matrix. The diagonal elements of $\Sigma$ are the singular values of $A$; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of $U$ and $V$ are the left and right singular vectors of $A$.

The user has to provide device and host working spaces which are pointed by input parameters $\text{bufferOnDevice}$ and $\text{bufferOnHost}$. The input parameters $\text{workspaceInBytesOnDevice}$ (and $\text{workspaceInBytesOnHost}$) is size in bytes of the device (and host) working space, and it is returned by $\text{cusolverDnXgesvd_bufferSize()}$.

If output parameter $\text{info} = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle). if $\text{bdsqr}$ did not converge, $\text{info}$ specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.

Currently, $\text{cusolverDnXgesvd}$ supports only the default algorithm.

**Table of algorithms supported by cusolverDnXgesvd**

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

Appendix G.5 provides a simple example of $\text{cusolverDnXgesvd}$.

Remark 1: $\text{gesvd}$ only supports $m \geq n$.

Remark 2: the routine returns $V^H$, not $V$.

List of input arguments for $\text{cusolverDnXgesvd_bufferSize}$ and $\text{cusolverDnXgesvd}$:

**API of cusolverDnXgesvd**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by $\text{cusolverDnSetAdvOptions}$.</td>
</tr>
<tr>
<td>jobu</td>
<td>host</td>
<td>input</td>
<td>specifies options for computing all or part of the matrix $U$: 'A': all $m$ columns of $U$ are returned in array $U$; 'S': the first $\min(m,n)$ columns of $U$ (the left singular vectors) are returned in the array $U$; 'O': the first $\min(m,n)$ columns of $U$ (the left singular vectors) are overwritten on the array $A$; 'N': no</td>
</tr>
</tbody>
</table>
columns of U [no left singular vectors] are computed.

<table>
<thead>
<tr>
<th>jobvt</th>
<th>host</th>
<th>input</th>
<th>specifies options for computing all or part of the matrix $V^*T$: = 'A': all $N$ rows of $V^*T$ are returned in the array VT; = 'S': the first $\min(m,n)$ rows of $V^*T$ [the right singular vectors] are returned in the array VT; = 'O': the first $\min(m,n)$ rows of $V^*T$ [the right singular vectors] are overwitten on the array $A$; = 'N': no rows of $V^*T$ [no right singular vectors] are computed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m$</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>$n$</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>input</td>
<td>data type of array $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension $lda * n$ with $lda$ is not less than $\max(1,m)$. On exit, the contents of $A$ are destroyed.</td>
</tr>
<tr>
<td>$lda$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>dataTypeS</td>
<td>host</td>
<td>input</td>
<td>data type of array $S$.</td>
</tr>
<tr>
<td>$S$</td>
<td>device</td>
<td>output</td>
<td>real array of dimension $\min(m,n)$. The singular values of $A$, sorted so that $S(i) &gt;= S(i+1)$.</td>
</tr>
<tr>
<td>dataTypeU</td>
<td>host</td>
<td>input</td>
<td>data type of array $U$.</td>
</tr>
<tr>
<td>$U$</td>
<td>device</td>
<td>output</td>
<td>array of dimension $ldu * m$ with $ldu$ is not less than $\max(1,m)$. $U$ contains the $m \times m$ unitary matrix $U$.</td>
</tr>
<tr>
<td>$ldu$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $U$.</td>
</tr>
<tr>
<td>dataTypeVT</td>
<td>host</td>
<td>input</td>
<td>data type of array $VT$.</td>
</tr>
<tr>
<td>$VT$</td>
<td>device</td>
<td>output</td>
<td>array of dimension $ldvt * n$ with $ldvt$ is not less than $\max(1,n)$. $VT$ contains the $n \times n$ unitary matrix $V^*T$.</td>
</tr>
<tr>
<td>$ldvt$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $VT$.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>bufferOnDevice</td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type void of size $workspaceInBytesOnDevice$ bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnDevice</td>
<td>host</td>
<td>input</td>
<td>size in bytes of $bufferOnDevice$, returned by $cusolverDnXpotrf_bufferSize$.</td>
</tr>
<tr>
<td>bufferOnHost</td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type void of size $workspaceInBytesOnHost$ bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnHost</td>
<td>host</td>
<td>input</td>
<td>size in bytes of $bufferOnHost$, returned by $cusolverDnXpotrf_bufferSize$.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>info</th>
<th>device</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>info = 0, the operation is successful.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if info = -i, the i-th parameter is wrong (not counting handle).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if info &gt; 0, info indicates how many superdiagonals of an intermediate bidiagonal form did not converge to zero.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The generic API has three different types, dataTypeA is data type of the matrix A, dataTypeS is data type of the vector S and dataTypeU is data type of the matrix U, dataTypeVT is data type of the matrix VT, computeType is compute type of the operation. cusolverDnXgesvd only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>DataTypeS</th>
<th>DataTypeU</th>
<th>DataTypeVT</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGESVD</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGESVD</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGESVD</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGESVD</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ([m, n&lt;0 \text{ or } \text{lda} &lt; \max(1, m) \text{ or } \text{ldu} &lt; \max(1, m) \text{ or } \text{ldvt} &lt; \max(1, n)}).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

**2.4.5.2. cusolverDnXgesvdp()**

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
int cusolverDnXgesvdp_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    int econ,
    int64_t m,
    int64_t n,
    cudaMemcpyDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaMemcpyDataType dataTypeS,
    const void *S,
    cudaMemcpyDataType dataTypeU,
    const void *U,
    int64_t ldu,
    cudaMemcpyDataType dataTypeV,
    const void *V,
    int64_t ldv,
    cudaDataType computeType,
    size_t *workspaceInBytesOnDevice,
    size_t *workspaceInBytesOnHost)
```
The routine below

```c
    cusolverStatus_t
cusolverDnXgesvdp(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    int econ,
    int64_t m,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType dataTypeS,
    void *S,
    cudaDataType dataTypeU,
    void *U,
    int64_t ldu,
    cudaDataType dataTypeV,
    void *V,
    int64_t ldv,
    cudaDataType computeType,
    void *bufferOnDevice,
    size_t workspaceInBytesOnDevice,
    void *bufferOnHost,
    size_t workspaceInBytesOnHost,
    int *d_info,
    double *h_err_sigma)
```

This function computes the singular value decomposition (SVD) of a \( m \times n \) matrix \( A \) and corresponding the left and/or right singular vectors. The SVD is written

\[
A = U \Sigma V^H
\]

where \( \Sigma \) is an \( m \times n \) matrix which is zero except for its \( \min(m, n) \) diagonal elements, \( U \) is an \( m \times m \) unitary matrix, and \( V \) is an \( n \times n \) unitary matrix. The diagonal elements of \( \Sigma \) are the singular values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m, n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

`cusolverDnXgesvdp` combines polar decomposition in [14] and `cusolverDnXsyevd` to compute SVD. It is much faster than `cusolverDnXgesvd` which is based on QR algorithm. However polar decomposition in [14] may not deliver a full unitary matrix when the matrix \( A \) has a singular value close to zero. To workaround the issue when the singular value is close to zero, we add a small perturbation so polar decomposition can deliver the correct result. The consequence is inaccurate singular values shifted by this perturbation. The output parameter `h_err_sigma` is the magnitude of this perturbation. In other words, `h_err_sigma` shows the accuracy of SVD.

The user has to provide device and host working spaces which are pointed by input parameters `bufferOnDevice` and `bufferOnHost`. The input parameters `workspaceInBytesOnDevice` (and `workspaceInBytesOnHost`) is size in bytes of the device (and host) working space, and it is returned by `cusolverDnXgesvdp_bufferSize()`.

If output parameter `info = -i` (less than zero), the \( i \)-th parameter is wrong (not counting handle).

Currently, `cusolverDnXgesvdp` supports only the default algorithm.

**Table of algorithms supported by cusolverDnXgesvdp**

| CUSOLVER_ALG_0 or NULL | Default algorithm. |
Appendix G.6 provides a simple example of cusolverDnXgesvdp.

Remark 1: gesvdp supports $n \geq m$ as well.

Remark 2: the routine returns $V$, not $V^H$.

List of input arguments for cusolverDnXgesvdp_bufferSize and cusolverDnXgesvdp:

**API of cusolverDnXgesvdp**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute singular values only or compute singular vectors as well $jobz = CUSOLVER_EIG_MODE_NOVECTOR$: Compute singular values only; $jobz = CUSOLVER_EIG_MODEVECTOR$: Compute singular values and singular vectors.</td>
</tr>
<tr>
<td>econ</td>
<td>host</td>
<td>input</td>
<td>$econ = 1$ for economy size for $U$ and $V$.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>input</td>
<td>data type of array $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>in/out</td>
<td>array of dimension $lda \times n$ with $lda$ is not less than $\max(1,m)$. On exit, the contents of $A$ are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A$.</td>
</tr>
<tr>
<td>dataTypeS</td>
<td>host</td>
<td>input</td>
<td>data type of array $S$.</td>
</tr>
<tr>
<td>$S$</td>
<td>device</td>
<td>output</td>
<td>real array of dimension $\min(m,n)$. The singular values of $A$, sorted so that $S(i) \geq S(i+1)$.</td>
</tr>
<tr>
<td>dataTypeU</td>
<td>host</td>
<td>input</td>
<td>data type of array $U$.</td>
</tr>
<tr>
<td>$U$</td>
<td>device</td>
<td>output</td>
<td>array of dimension $ldu \times m$ with $ldu$ is not less than $\max(1,m)$. $U$ contains the $m \times m$ unitary matrix $U$. if $econ=1$, only reports first $\min(m,n)$ columns of $U$.</td>
</tr>
<tr>
<td>ldu</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $U$.</td>
</tr>
<tr>
<td>dataTypeV</td>
<td>host</td>
<td>input</td>
<td>data type of array $V$.</td>
</tr>
<tr>
<td>$V$</td>
<td>device</td>
<td>output</td>
<td>array of dimension $ldv \times n$ with $ldv$ is not less than $\max(1,n)$. $V$ contains the $n \times n$ unitary matrix $V$. if $econ=1$, only reports first $\min(m,n)$ columns of $V$.</td>
</tr>
<tr>
<td>ldv</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $V$.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>data type of computation.</td>
</tr>
</tbody>
</table>
bufferOnDevice | device | in/out | device workspace. Array of type void of size workspaceInBytesOnDevice bytes.
--- | --- | --- | ---
workspaceInBytesOnDevice | input | size in bytes of bufferOnDevice, returned by cusolverDnXpotrf_bufferSize.
bufferOnHost | device | in/out | host workspace. Array of type void of size workspaceInBytesOnHost bytes.
workspaceInBytesOnHost | input | size in bytes of bufferOnHost, returned by cusolverDnXpotrf_bufferSize.
info | device | output | if info = 0, the operation is successful. if info = -i, the i-th parameter is wrong (not counting handle).

The generic API has three different types, dataTypeA is data type of the matrix A, dataTypeS is data type of the vector S and dataTypeU is data type of the matrix U, dataTypeV is data type of the matrix V, computeType is compute type of the operation. cusolverDnXgesvdp only supports the following four combinations.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>DataTypeS</th>
<th>DataTypeU</th>
<th>DataTypeV</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SGESVDP</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DGESVDP</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CGESVDP</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZGESVDP</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( m, n \leq 0 ) or ( lda &lt; \max(1, m) ) or ( ldu &lt; \max(1, m) ) or ( ldv &lt; \max(1, n) ).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.4.5.3. cusolverDnXsyevd()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

cusolverStatus_t

```c

cusolverDnXsyevd_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int64_t n,
    cudaDataType dataTypeA,
    const void *A,
    int64_t lda,
    cudaDataType dataTypeW,
    const void *W,
    cudaDataType computeType,
    size_t *workspaceInBytesOnDevice,
```
The routine bellow

```
cusolverStatus_t
cusolverDnXsyevd(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int64_t n,
    cudaDataType dataTypeA,
    void *A,
    int64_t lda,
    cudaDataType dataTypeW,
    void *W,
    cudaDataType computeType,
    void *bufferOnDevice,
    size_t workspaceInBytesOnDevice,
    void *bufferOnHost,
    size_t workspaceInBytesOnHost,
    int *info)
```

computes eigenvalues and eigenvectors of a symmetric (Hermitian) \( n \times n \) matrix \( A \) using the generic API interface. The standard symmetric eigenvalue problem is

\[
A V = V \Lambda
\]

where \( \Lambda \) is a real \( n \times n \) diagonal matrix. \( V \) is an \( n \times n \) unitary matrix. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

The user has to provide device and host working spaces which are pointed by input parameters `bufferOnDevice` and `bufferOnHost`. The input parameters `workspaceInBytesOnDevice` (and `workspaceInBytesOnHost`) is size in bytes of the device (and host) working space, and it is returned by `cusolverDnXsyevd_bufferSize()`.

If output parameter `info = -i` (less than zero), the \( i \)-th parameter is wrong (not counting handle). If `info = i` (greater than zero), \( i \) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If `jobz = CUSOLVER_EIG_MODE_VECTOR`, \( A \) contains the orthonormal eigenvectors of the matrix \( \Lambda \). The eigenvectors are computed by a divide and conquer algorithm.

Currently, `cusolverDnXsyevd` supports only the default algorithm.

### Table of algorithms supported by `cusolverDnXsyevd`

<table>
<thead>
<tr>
<th>CUSOLVER_ALG_0 or NULL</th>
<th>Default algorithm.</th>
</tr>
</thead>
</table>

### List of input arguments for `cusolverDnXsyevd_bufferSize` and `cusolverDnXsyevd`:

#### API of `cusolverDnXsyevd`

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by <code>cusolverDnSetAdvOptions</code>.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: <code>jobz = CUSOLVER_EIG_MODE_NOVECTOR</code></td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jobz</code></td>
<td></td>
<td></td>
<td>: Compute eigenvalues only; <code>jobz = CUSOLVER_EIG_MODE_VECTOR</code> : Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td><code>uplo</code></td>
<td>host</td>
<td>input</td>
<td>specifies which part of ( A ) is stored. <code>uplo = CUBLAS_FILL_MODE_LOWER</code>: Lower triangle of ( A ) is stored. <code>uplo = CUBLAS_FILL_MODE_UPPER</code>: Upper triangle of ( A ) is stored.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix ( A ).</td>
</tr>
<tr>
<td><code>dataTypeA</code></td>
<td>host</td>
<td>in</td>
<td>data type of array ( A ).</td>
</tr>
<tr>
<td><code>A</code></td>
<td>device</td>
<td>in/out</td>
<td>array of dimension ( lda \times n ) with ( lda ) is not less than ( \max(1,n) ). If <code>uplo = CUBLAS_FILL_MODE_UPPER</code>, the leading ( n )-by-( n ) upper triangular part of ( A ) contains the upper triangular part of the matrix ( A ). If <code>uplo = CUBLAS_FILL_MODE_LOWER</code>, the leading ( n )-by-( n ) lower triangular part of ( A ) contains the lower triangular part of the matrix ( A ). On exit, if <code>jobz = CUSOLVER_EIG_MODE_VECTOR</code>, and <code>info = 0</code>, ( A ) contains the orthonormal eigenvectors of the matrix ( A ). If <code>jobz = CUSOLVER_EIG_MODE_NOVECTOR</code>, the contents of ( A ) are destroyed.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( A ).</td>
</tr>
<tr>
<td><code>dataTypeW</code></td>
<td>host</td>
<td>in</td>
<td>data type of array ( W ).</td>
</tr>
<tr>
<td><code>W</code></td>
<td>device</td>
<td>output</td>
<td>a real array of dimension ( n ). The eigenvalue values of ( A ), in ascending order ie, sorted so that ( W(i) &lt;= W(i+1) ).</td>
</tr>
<tr>
<td><code>computeType</code></td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td><code>bufferOnDevice</code></td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type <code>void</code> of size ( \text{workspaceInBytesOnDevice} ) bytes.</td>
</tr>
<tr>
<td><code>workspaceInBytesDevice</code></td>
<td>input</td>
<td></td>
<td>size in bytes of bufferOnDevice, returned by <code>cusolverDnXpotrf_bufferSize</code>.</td>
</tr>
<tr>
<td><code>bufferOnHost</code></td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type <code>void</code> of size ( \text{workspaceInBytesOnHost} ) bytes.</td>
</tr>
<tr>
<td><code>workspaceInBytesHost</code></td>
<td>input</td>
<td></td>
<td>size in bytes of bufferOnHost, returned by <code>cusolverDnXpotrf_bufferSize</code>.</td>
</tr>
<tr>
<td><code>info</code></td>
<td>device</td>
<td>output</td>
<td>if <code>info = 0</code>, the operation is successful. if <code>info = -i</code>, the ( i )-th parameter is wrong (not counting handle). if <code>info = i ( &gt; 0)</code>, <code>info</code> indicates ( i ) off-diagonal elements of an intermediate tridiagonal form did not converge to zero;</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

The generic API has three different types, `dataTypeA` is data type of the matrix `A`, `dataTypeW` is data type of the matrix `W` and `computeType` is compute type of the operation. `cusolverDnXsyevd` only supports the following four combinations.

### Valid combination of data type and compute type

<table>
<thead>
<tr>
<th><code>DataTypeA</code></th>
<th><code>DataTypeW</code></th>
<th><code>ComputeType</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SSYEVD</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DSYEVD</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CHEEVD</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZHEEVD</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (<code>n&lt;0</code>, or <code>lda&lt;max(1,n)</code>), or <code>jobz</code> is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or <code>uplo</code> is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.4.5.4. `cusolverDnXsyevdx()`

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverDnXsyevdx_bufferSize(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int64_t n,
    cusDataType dataTypeA,
    const void *A,
    void *vl,
    void *vu,
    int64_t il,
    int64_t iu,
    int64_t *h_meig,
    cusDataType dataTypeW,
    const void *W,
    cusDataType computeType,
    size_t *workspaceInBytesOnDevice,
    size_t *workspaceInBytesOnHost)
```

The routine bellow

```c
cusolverStatus_t CUSOLVERAPI cusolverDnXsyevdx(
    cusolverDnHandle_t handle,
    cusolverDnParams_t params,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
```
computes all or selection of the eigenvalues and optionally eigenvectors of a symmetric (Hermitian) $n \times n$ matrix $\Lambda$ using the generic API interface. The standard symmetric eigenvalue problem is

$$A \ast V = V \ast \Lambda$$

where $\Lambda$ is a real $n \times h_{meig}$ diagonal matrix. $V$ is an $n \times h_{meig}$ unitary matrix. $h_{meig}$ is the number of eigenvalues/eigenvectors computed by the routine. $h_{meig}$ is equal to $n$ when the whole spectrum (e.g., range = CUSOLVER_EIG_RANGE_ALL) is requested. The diagonal elements of $\Lambda$ are the eigenvalues of $A$ in ascending order.

The user has to provide device and host working spaces which are pointed by input parameters bufferOnDevice and bufferOnHost. The input parameters workspaceInBytesOnDevice (and workspaceInBytesOnHost) is size in bytes of the device (and host) working space, and it is returned by cusolverDnXsyevdx_bufferSize().

If output parameter info = -i (less than zero), the i-th parameter is wrong (not counting handle). If info = i (greater than zero), $i$ off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

if jobz = CUSOLVER_EIG_MODE_VECTOR, $A$ contains the orthonormal eigenvectors of the matrix $\Lambda$. The eigenvectors are computed by a divide and conquer algorithm.

Currently, cusolverDnXsyevdx supports only the default algorithm.

**Table of algorithms supported by cusolverDnXsyevdx**

| CUSOLVER_ALG_0 or NULL | Default algorithm. |

**List of input arguments for cusolverDnXsyevdx_bufferSize and cusolverDnXsyevdx:**

**API of cusolverDnXsyevdx**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>params</td>
<td>host</td>
<td>input</td>
<td>structure with information collected by cusolverDnSetAdvOptions.</td>
</tr>
</tbody>
</table>
| jobz      | host   | input  | specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; jobz =  

```c
int64_t n,
cudaDataType dataTypeA,
void *A,
int64_t lda,
void * vl,
void * vu,
int64_t ii,
int64_t iiu,
int64_t *meig64,
cudaDataType dataTypeW,
void *W,
cudaDataType computeType,
void *bufferOnDevice,
size_t workspaceInBytesOnDevice,
void *bufferOnHost,
size_t workspaceInBytesOnHost,
int *info)
```
CUSOLVER_EIG_MODE_VECTOR : Compute eigenvalues and eigenvectors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: range = CUSOLVER_EIG_RANGE_ALL : all eigenvalues/eigenvectors will be found, will becomes the classical syevd/heevd routine; range = CUSOLVER_EIG_RANGE_V : all eigenvalues/eigenvectors in the half-open interval (vl,vu] will be found; range = CUSOLVER_EIG_RANGE_I : the il-th through iu-th eigenvalues/eigenvectors will be found;</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specifies which part of A is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of rows [or columns] of matrix A.</td>
</tr>
<tr>
<td>dataTypeA</td>
<td>host</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data type of array A.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
</tr>
<tr>
<td></td>
<td></td>
<td>array of dimension lda * n with lda is not less than max(1,n). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A. On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info = 0, A contains the orthonormal eigenvectors of the matrix A. If jobz = CUSOLVER_EIG_MODE_NOVECTOR, the contents of A are destroyed.</td>
</tr>
<tr>
<td>lda</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>leading dimension of two-dimensional array used to store matrix A. lda is not less than max(1,n).</td>
</tr>
<tr>
<td>vl, vu</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If range = CUSOLVER_EIG_RANGE_V, the lower and upper bounds of the interval to be searched for eigenvalues. vl &gt; vu. Not referenced if range = CUSOLVER_EIG_RANGE_ALL or range = CUSOLVER_EIG_RANGE_I. Note that, if eigenvalues are very close to each other, it is well known that two different eigenvalues routines might find slightly different number of eigenvalues inside the same interval. This is due to the fact</td>
</tr>
</tbody>
</table>
that different eigenvalue algorithms, or even same algorithm but different run might find eigenvalues within some rounding error close to the machine precision. Thus, if the user want to be sure not to miss any eigenvalue within the interval bound, we suggest that, the user substract/add epsilon (machine precision) to the interval bound such as \([vl=vl-\text{eps}, vu=vu+\text{eps}]\). This suggestion is valid for any selective routine from cuSolver or LAPACK.

<table>
<thead>
<tr>
<th>il, iu</th>
<th>host</th>
<th>input</th>
<th>integer. If ( \text{range} = \text{CUSOLVER_EIG_RANGE_I} ), the indices (in ascending order) of the smallest and largest eigenvalues to be returned. (1 \leq il \leq iu \leq n), if ( n &gt; 0 ); ( il = 1 ) and ( iu = 0 ) if ( n = 0 ). Not referenced if ( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} ) or ( \text{range} = \text{CUSOLVER_EIG_RANGE_V} ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_meig</td>
<td>host</td>
<td>output</td>
<td>integer. The total number of eigenvalues found. ( 0 \leq h_meig \leq n ). If ( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} ), ( h_meig = n ), and if ( \text{range} = \text{CUSOLVER_EIG_RANGE_I} ), ( h_meig = iu-il+1 ).</td>
</tr>
<tr>
<td>dataTypeW</td>
<td>host</td>
<td>in</td>
<td>data type of array ( W ).</td>
</tr>
<tr>
<td>W</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension ( n ). The eigenvalue values of ( A ), in ascending order ie, sorted so that ( W(i) \leq W(i+1) ).</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>in</td>
<td>data type of computation.</td>
</tr>
<tr>
<td>bufferOnDevice</td>
<td>device</td>
<td>in/out</td>
<td>device workspace. Array of type void of size workspaceInBytesOnDevice bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnDevice</td>
<td>input</td>
<td>size in bytes of bufferOnDevice, returned by cusolverDnXpotrf_bufferSize.</td>
<td></td>
</tr>
<tr>
<td>bufferOnHost</td>
<td>device</td>
<td>in/out</td>
<td>host workspace. Array of type void of size workspaceInBytesOnHost bytes.</td>
</tr>
<tr>
<td>workspaceInBytesOnHost</td>
<td>input</td>
<td>size in bytes of bufferOnHost, returned by cusolverDnXpotrf_bufferSize.</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if ( \text{info} = 0 ), the operation is successful. If ( \text{info} = -i ), the ( i )-th parameter is wrong (not counting handle). If ( \text{info} = i (&gt; 0) ), info indicates ( i ) off-diagonal elements of an intermediate tridiagonal form did not converge to zero;</td>
</tr>
</tbody>
</table>

The generic API has three different types, \( \text{dataTypeA} \) is data type of the matrix \( A \), \( \text{dataTypeW} \) is data type of the matrix \( W \) and \( \text{computeType} \) is compute type of the operation. \text{cusolverDnXsyevdx} only supports the following four combinations.
Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>DataTypeW</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SSYEVDX</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DSYEVDX</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CHEEVDX</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZHEEVDX</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( n &lt; 0 ), ( l d a &lt; \max(1, n) ), or ( j o b z ) is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or range is not CUSOLVER_EIG_RANGE_ALL or CUSOLVER_EIG_RANGE_V or CUSOLVER_EIG_RANGE_I, or uplo is not CUBLAS_FILL_MODE_LOWER or CUBLAS_FILL_MODE_UPPER.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

2.5. cuSolverSP: sparse LAPACK Function Reference

This chapter describes the API of cuSolverSP, which provides a subset of LAPACK functions for sparse matrices in CSR or CSC format.

2.5.1. Helper Function Reference

2.5.1.1. cusolverSpCreate()

cusolverStatus_t
cusolverSpCreate(cusolverSpHandle_t *handle)

This function initializes the cuSolverSP library and creates a handle on the cuSolver context. It must be called before any other cuSolverSP API function is invoked. It allocates hardware resources necessary for accessing the GPU.

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>the pointer to the handle to the cuSolverSP context.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the initialization succeeded.</td>
</tr>
</tbody>
</table>
CUSOLVER_STATUS_NOT_INITIALIZED | the CUDA Runtime initialization failed.
CUSOLVER_STATUS_ALLOC_FAILED | the resources could not be allocated.
CUSOLVER_STATUS_ARCH_MISMATCH | the device only supports compute capability 2.0 and above.

### 2.5.1.2. cusolverSpDestroy()

```c
 cusolverStatus_t
cusolverSpDestroy(cusolverSpHandle_t handle)
```

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

**Input**

| handle | the handle to the cuSolverSP context. |

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the shutdown succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

### 2.5.1.3. cusolverSpSetStream()

```c
 cusolverStatus_t
cusolverSpSetStream(cusolverSpHandle_t handle, cudaStream_t streamId)
```

This function sets the stream to be used by the cuSolverSP library to execute its routines.

**Input**

| handle | the handle to the cuSolverSP context. |
| streamId | the stream to be used by the library. |

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the stream was set successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

### 2.5.1.4. cusolverSpXcsrissym()

```c
 cusolverStatus_t
cusolverSpXcsrissymHost(cusolverSpHandle_t handle, int m, int nnzA, const cusparseMatDescr_t descrA, const int *csrRowPtrA, const int *csrEndPtrA, const int *csrColIndA, int *issym);
```

This function checks if $A$ has symmetric pattern or not. The output parameter `issym` reports 1 if $A$ is symmetric; otherwise, it reports 0.
The matrix $A$ is an $m \times m$ sparse matrix that is defined in CSR storage format by the four arrays $\text{csrValA}$, $\text{csrRowPtrA}$, $\text{csrEndPtrA}$ and $\text{csrColIndA}$.

The supported matrix type is $\text{CUSPARSE_MATRIX_TYPE_GENERAL}$.

The $\text{csrlsvlu}$ and $\text{csrlsvqr}$ do not accept non-general matrix. the user has to extend the matrix into its missing upper/lower part, otherwise the result is not expected. The user can use $\text{csrissym}$ to check if the matrix has symmetric pattern or not.

Remark 1: only CPU path is provided.

Remark 2: the user has to check returned status to get valid information. The function converts $A$ to CSC format and compare CSR and CSC format. If the CSC failed because of insufficient resources, $\text{issym}$ is undefined, and this state can only be detected by the return status code.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemorySpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>$m$</td>
<td>host</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td>$\text{nnzA}$</td>
<td>host</td>
<td>number of nonzeros of matrix $A$. It is the size of $\text{csrValA}$ and $\text{csrColIndA}$.</td>
</tr>
<tr>
<td>$\text{descrA}$</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is $\text{CUSPARSE_MATRIX_TYPE_GENERAL}$. Also, the supported index bases are $\text{CUSPARSE_INDEX_BASE_ZERO}$ and $\text{CUSPARSE_INDEX_BASE_ONE}$.</td>
</tr>
<tr>
<td>$\text{csrRowPtrA}$</td>
<td>host</td>
<td>integer array of $m$ elements that contains the start of every row.</td>
</tr>
<tr>
<td>$\text{csrEndPtrA}$</td>
<td>host</td>
<td>integer array of $m$ elements that contains the end of the last row plus one.</td>
</tr>
<tr>
<td>$\text{csrColIndA}$</td>
<td>host</td>
<td>integer array of $\text{nnzA}$ column indices of the nonzero elements of matrix $A$.</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemorySpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{issym}$</td>
<td>host</td>
<td>1 if $A$ is symmetric; 0 otherwise.</td>
</tr>
</tbody>
</table>

Status Returned

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_ALLOC_FAILED**: the resources could not be allocated.
- **CUSOLVER_STATUS_INVALID_VALUE**: invalid parameters were passed ($m, \text{nnzA} \leq 0$), base index is not 0 or 1.
- **CUSOLVER_STATUS_ARCH_MISMATCH**: the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.
- **CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED**: the matrix type is not supported.
2.5.2. High Level Function Reference

This section describes high level API of cuSolverSP, including linear solver, least-square solver and eigenvalue solver. The high-level API is designed for ease-of-use, so it allocates any required memory under the hood automatically. If the host or GPU system memory is not enough, an error is returned.

2.5.2.1. cusolverSp<t>csrlsvlu()

cusolverStatus_t
cusolverSpScsrlsvlu[Host](cusolverSpHandle_t handle,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const float *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
f
const float *b,
f
float tol,
int reorder,
f
float *x,
f
int *singularity);

cusolverStatus_t
cusolverSpDcsrlsvlu[Host](cusolverSpHandle_t handle,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const double *b,
double tol,
int reorder,
double *x,
int *singularity);

cusolverStatus_t
cusolverSpCcsrlsvlu[Host](cusolverSpHandle_t handle,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const cuComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const cuComplex *b,
f
float tol,
int reorder,
cuComplex *x,
int *singularity);

cusolverStatus_t
cusolverSpZcsrlsvlu[Host](cusolverSpHandle_t handle,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
This function solves the linear system

\[ A x = b \]

\( A \) is an \( n \times n \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \). \( b \) is the right-hand-side vector of size \( n \), and \( x \) is the solution vector of size \( n \).

The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. If matrix \( A \) is symmetric/Hermitian and only lower/upper part is used or meaningful, the user has to extend the matrix into its missing upper/lower part, otherwise the result would be wrong.

The linear system is solved by sparse LU with partial pivoting,

\[ P^* A = L^* U \]

\text{cusolver} library provides three reordering schemes, \text{symrcm} \, \text{symamd}, and \text{csrmetsisnd} to reduce zero fill-in which dramatically affects the performance of LU factorization. The input parameter \( \text{reorder} \) can enable \text{symrcm} \, \text{symamd} or \text{csrmetsisnd} if \( \text{reorder} \) is \( 1 \) \, \( 2 \), or \( 3 \), otherwise, no reordering is performed.

If \( \text{reorder} \) is nonzero, \text{csrlsvlu} does

\[ P^* A^* Q^T = L^* U \]

where \( Q = \text{symrcm}(A + A^T) \).

If \( A \) is singular under given tolerance \( \max(\text{tol}, 0) \), then some diagonal elements of \( U \) is zero, i.e.

\[ |U(j,j)| < \text{tol} \text{ for some } j \]

The output parameter \( \text{singularity} \) is the smallest index of such \( j \). If \( A \) is non-singular, \( \text{singularity} \) is \( -1 \). The index is base-0, independent of base index of \( A \). For example, if 2nd column of \( A \) is the same as first column, then \( A \) is singular and \( \text{singularity} = 1 \) which means \( U(1,1) \approx 0 \).

Remark 1: \text{csrlsvlu} performs traditional LU with partial pivoting, the pivot of \( k \)-th column is determined dynamically based on the \( k \)-th column of intermediate matrix. \text{csrlsvlu} follows Gilbert and Peierls’s algorithm [4] which uses depth-first-search and topological ordering to solve triangular system [Davis also describes this algorithm in detail in his book [1]]. since \text{cuda 10.1}, \text{csrlsvlu} will incrementally reallocate the memory to store \( L \) and \( U \). This feature can avoid over-estimate size from QR factorization. In some cases, zero fill-in of QR can be order of magnitude higher than LU.

Remark 2: only CPU (Host) path is provided.

Input

```c
const int *csrRowPtrA,
const int *csrColIndA,
const cuDoubleComplex *b,
double tol,
int reorder,
cuDoublesComplex *x,
int *singularity);
```
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*HostMemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>(&lt;\text{type}&gt;\text{ array of } \text{nnzA} = \text{csrRowPtrA}(n) - \text{csrRowPtrA}(0)) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of ( n + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>host</td>
<td>integer array of ( \text{nnzA} = \text{csrRowPtrA}(n) - \text{csrRowPtrA}(0)) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>b</td>
<td>device</td>
<td>host</td>
<td>right hand side vector of size ( n ).</td>
</tr>
<tr>
<td>tol</td>
<td>host</td>
<td>host</td>
<td>tolerance to decide if singular or not.</td>
</tr>
<tr>
<td>reorder</td>
<td>host</td>
<td>host</td>
<td>no ordering if ( \text{reorder}=0 ). Otherwise, ( \text{symrcm}, \text{symamd}, \text{or csrmetisnd} ) is used to reduce zero fill-in.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*HostMemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>device</td>
<td>host</td>
<td>solution vector of size ( n ), ( x = \text{inv}(A)\text{b} ).</td>
</tr>
<tr>
<td>singularity</td>
<td>host</td>
<td>host</td>
<td>-1 if ( A ) is invertible. Otherwise, first index ( j ) such that ( U(j,j) \approx 0 ).</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ( [n, \text{nnzA}&lt;=0, \text{base} \text{index is not } 0 \text{ or } 1, \text{reorder} \text{is not } 0,1,2,3] )</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>
2.5.2.2. cusolverSp<t>csrlsvqr()

cusolverStatus_t
cusolverSpScsrIsvqr[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const float *b,
    float tol,
    int reorder,
    float *x,
    int *singularity);

cusolverStatus_t
cusolverSpDcsrlsvqr[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const double *b,
    double tol,
    int reorder,
    double *x,
    int *singularity);

cusolverStatus_t
cusolverSpCcsrlsvqr[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cuComplex *b,
    float tol,
    int reorder,
    cuComplex *x,
    int *singularity);

cusolverStatus_t
cusolverSpZcsrlsvqr[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cuDoubleComplex *b,
    double tol,
    int reorder,
    cuDoubleComplex *x,
    int *singularity);
This function solves the linear system

\[ A\mathbf{x} = \mathbf{b} \]

\(A\) is an \(m \times m\) sparse matrix that is defined in CSR storage format by the three arrays \(\text{csrValA}\), \(\text{csrRowPtrA}\), and \(\text{csrColIndA}\). \(\mathbf{b}\) is the right-hand-side vector of size \(m\), and \(\mathbf{x}\) is the solution vector of size \(m\).

The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}. If matrix \(A\) is symmetric/Hermitian and only lower/upper part is used or meaningful, the user has to extend the matrix into its missing upper/lower part, otherwise the result would be wrong.

The linear system is solved by sparse QR factorization,

\[ A = QR \]

If \(A\) is singular under given tolerance \(\max(\text{tol}, 0)\), then some diagonal elements of \(R\) is zero, i.e.

\[ |R(j,j)| < \text{tol} \text{ for some } j \]

The output parameter \(\text{singularity}\) is the smallest index of such \(j\). If \(A\) is non-singular, \(\text{singularity}\) is \(-1\). The \(\text{singularity}\) is base-0, independent of base index of \(A\). For example, if 2nd column of \(A\) is the same as first column, then \(A\) is singular and \(\text{singularity} = 1\) which means \(R(1,1) \approx 0\).

cusolver library provides three reordering schemes, \texttt{symrcm} \texttt{symamd}, and \texttt{csrmetisnd} to reduce zero fill-in which dramatically affects the performance of QR factorization. The input parameter \(\text{reorder}\) can enable \texttt{symrcm} \{\texttt{symamd} or \texttt{csrmetisnd}\} if \(\text{reorder}\) is \(1\) \{\(2\), or \(3\), otherwise, no reordering is performed.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*HostMemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cusolverSP library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix (A).</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix (A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix (A). The supported matrix type is \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are \texttt{CUSPARSE_INDEX_BASE_ZERO} and \texttt{CUSPARSE_INDEX_BASE_ONE}.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>&lt;type&gt; array of (\text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) nonzero elements of matrix (A).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of (m + 1) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>host</td>
<td>integer array of (\text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) column indices of the nonzero elements of matrix (A).</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>b</th>
<th>device</th>
<th>host</th>
<th>right hand side vector of size m.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tol</td>
<td>host</td>
<td>host</td>
<td>tolerance to decide if singular or not.</td>
</tr>
<tr>
<td>reorder</td>
<td>host</td>
<td>host</td>
<td>no ordering if reorder=0. Otherwise, symrcm, symamd, or csrmetisnd is used to reduce zero fill-in.</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>device</td>
<td>host</td>
<td>solution vector of size m, x = inv(A)*b.</td>
</tr>
<tr>
<td>singularity</td>
<td>host</td>
<td>host</td>
<td>-1 if A is invertible. Otherwise, first index j such that R(j,j) ≈ 0</td>
</tr>
</tbody>
</table>

Status Returned

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_ALLOC_FAILED: the resources could not be allocated.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed (m, nnz<=0, base index is not 0 or 1, reorder is not 0,1,2,3)
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.
- CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED: the matrix type is not supported.

2.5.2.3. cusolverSp<t>csrlsvchol()

cusolverStatus_t
cusolverSpScsrlsvchol[Host](cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const float *csrVal,
const int *csrRowPtr,
const int *csrColInd,
const float *b,
float tol,
int reorder,
float *x,
int *singularity);

cusolverStatus_t
cusolverSpDcsrlsvchol[Host](cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const double *csrVal,
const int *csrRowPtr,
const int *csrColInd,
const double *b,
double tol,
This function solves the linear system

\[ A x = b \]

\( A \) is an \( m \times m \) symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \). \( b \) is the right-hand-side vector of size \( m \), and \( x \) is the solution vector of size \( m \).

The supported matrix type is \text{CUSPARSE\_MATRIX\_TYPE\_GENERAL} and upper triangular part of \( A \) is ignored (if parameter \( \text{reorder} \) is zero). In other words, suppose input matrix \( A \) is decomposed as \( A = L + D + U \), where \( L \) is lower triangular, \( D \) is diagonal and \( U \) is upper triangular. The function would ignore \( U \) and regard \( A \) as a symmetric matrix with the formula \( A = L + D + L^H \). If parameter \( \text{reorder} \) is nonzero, the user has to extend \( A \) to a full matrix, otherwise the solution would be wrong.

The linear system is solved by sparse Cholesky factorization,

\[ A = G^*G^H \]

where \( G \) is the Cholesky factor, a lower triangular matrix.

The output parameter \text{singularity} has two meanings:

- If \( A \) is not positive definite, there exists some integer \( k \) such that \( A(0:k, 0:k) \) is not positive definite. \text{singularity} is the minimum of such \( k \).
If $A$ is positive definite but near singular under tolerance $\max(\text{tol}, 0)$, i.e. there exists some integer $k$ such that $G(k, k) < \text{tol}$. singularity is the minimum of such $k$.

singularity is base-0. If $A$ is positive definite and not near singular under tolerance, singularity is -1. If the user wants to know if $A$ is positive definite or not, $\text{tol}=0$ is enough.

cusolver library provides three reordering schemes, symrcm, symamd, and csrmetisnd to reduce zero fill-in which dramatically affects the performance of Cholesky factorization. The input parameter reorder can enable symrcm (symamd or csrmetisnd) if reorder is 1 (2, or 3), otherwise, no reordering is performed.

Remark 1: the function works for in-place ($x$ and $b$ point to the same memory block) and out-of-place.

Remark 2: the function only works on 32-bit index, if matrix $G$ has large zero fill-in such that number of nonzeros is bigger than $2^{31}$, then CUSOLVER_STATUS_ALLOC_FAILED is returned.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>$\langle$type$\rangle$ array of $\text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of $m + 1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>host</td>
<td>integer array of $\text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ column indices of the nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>b</td>
<td>device</td>
<td>host</td>
<td>right hand side vector of size $m$.</td>
</tr>
<tr>
<td>tol</td>
<td>host</td>
<td>host</td>
<td>tolerance to decide singularity.</td>
</tr>
<tr>
<td>reorder</td>
<td>host</td>
<td>host</td>
<td>no ordering if reorder=0. Otherwise, symrcm, symamd, or csrmetisnd is used to reduce zero fill-in.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>device</td>
<td>host</td>
<td>solution vector of size $m$, $x = \text{inv}(A) \cdot b$.</td>
</tr>
<tr>
<td>singularity</td>
<td>host</td>
<td>host</td>
<td>-1 if $A$ is symmetric positive definite.</td>
</tr>
</tbody>
</table>
Status Returned

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (m, nnz&lt;=0, base index is not 0 or 1, reorder is not 0,1,2,3)</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>

2.5.2.4. cusolverSp<t>csrlsqvqr()

The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverSpScsrlsqvqr[Host](cusolverSpHandle_t handle,
int m,
int n,
ext nnz,
const cusparseMatDescr_t descrA,
const float *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const float *b,
float tol,
int *rankA,
float *x,
int *p,
float *min_norm);

cusolverStatus_t
cusolverSpDcsrlsqvqr[Host](cusolverSpHandle_t handle,
int m,
int n,
ext nnz,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const double *b,
double tol,
int *rankA,
double *x,
int *p,
double *min_norm);

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverSpCcsrlsqvqr[Host](cusolverSpHandle_t handle,
int m,
int n,
ext nnz,
const cusparseMatDescr_t descrA,
const cuComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const cuComplex *b,
float tol,
int *rankA,
cuComplex *x,
int *p,
float *min_norm);

cusolverStatus_t
cusolverSpZcsrlsqvqr[Host](cusolverSpHandle_t handle,
int m,
int n,
nint nnz,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const cuDoubleComplex *b,
double tol,
int *rankA,
cuDoubleComplex *x,
int *p,
double *min_norm);

This function solves the following least-square problem

$$x = \text{argmin}\|A^*z - b\|$$

$A$ is an $m \times n$ sparse matrix that is defined in CSR storage format by the three arrays $\text{csrValA}$, $\text{csrRowPtrA}$, and $\text{csrColIndA}$. $b$ is the right-hand-side vector of size $m$, and $x$ is the least-square solution vector of size $n$.

The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. If $A$ is square, symmetric/Hermitian and only lower/upper part is used or meaningful, the user has to extend the matrix into its missing upper/lower part, otherwise the result is wrong.

This function only works if $m$ is greater or equal to $n$, in other words, $A$ is a tall matrix.

The least-square problem is solved by sparse QR factorization with column pivoting,

$$A^*P^T = Q*R$$

If $A$ is of full rank (i.e. all columns of $A$ are linear independent), then matrix $P$ is an identity. Suppose rank of $A$ is $k$, less than $n$, the permutation matrix $P$ reorders columns of $A$ in the following sense:

$$A^*P^T = (A_1 \ A_2) = (Q_1 \ Q_2) \begin{bmatrix}R_{11} & R_{12} \\ R_{21} & R_{22}\end{bmatrix}$$

where $R_{11}$ and $A$ have the same rank, but $R_{22}$ is almost zero, i.e. every column of $A_2$ is linear combination of $A_1$.

The input parameter $tol$ decides numerical rank. The absolute value of every entry in $R_{22}$ is less than or equal to $\text{tolerance}=\max(\text{tol}, 0)$.

The output parameter $\text{rankA}$ denotes numerical rank of $A$. 

Suppose $y = P^*x$ and $c = Q^H* b$, the least square problem can be reformed by
\[
\min \|A^*x - b\| = \min \|R^*y - c\|
\]
or in matrix form
\[
\begin{pmatrix}
R_{11} & R_{12} \\
R_{21} & R_{22}
\end{pmatrix}
\begin{pmatrix}
y_1 \\
y_2
\end{pmatrix}
= \begin{pmatrix} c_1 \\
c_2
\end{pmatrix}
\]
The output parameter $\min_{\text{norm}}$ is $\|c_2\|$, which is minimum value of least-square problem.

If $A$ is not of full rank, above equation does not have a unique solution. The least-square problem is equivalent to
\[
\min \|y\| \\
\text{subject to } R_{11}^*y_1 + R_{12}^*y_2 = c_1
\]
Or equivalently another least-square problem
\[
\min \| \begin{pmatrix} R_{11} \backslash R_{12} \\
I \\
O
\end{pmatrix} y_2 - \begin{pmatrix} R_{11} \backslash c_1 \\
O
\end{pmatrix} \| 
\]
The output parameter $x$ is $P^T* y$, the solution of least-square problem.

The output parameter $p$ is a vector of size $n$. It corresponds to a permutation matrix $P$. $p(i) = j$ means $(P^*x)(i) = x(j)$. If $A$ is of full rank, $p = 0:n-1$.

Remark 1: $p$ is always base 0, independent of base index of $A$.

Remark 2: only CPU (Host) path is provided.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolver library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>host</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>host</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>&lt;type&gt; array of nnz (= csrRowPtrA(m) - csrRowPtrA(0)) nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of $m + 1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

### csrColIndA
- **device**
- **host**
- integer array of `nnz (= csrRowPtrA(m) - csrRowPtrA(0))` column indices of the nonzero elements of matrix `A`.

### b
- **device**
- **host**
- right hand side vector of size `m`.

### tol
- **host**
- **host**
- tolerance to decide rank of `A`.

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
</table>
| rankA     | host               | host           | numerical rank of `A`.
| x         | device             | host           | solution vector of size `n`, `x=pinv(A)*b`.
| p         | device             | host           | a vector of size `n`, which represents the permutation matrix `P` satisfying `A*P^T=Q*R`.
| min_norm  | host               | host           | `||A*x-b||`, `x=pinv(A)*b`.

### Status Returned

- **CUSOLVER_STATUS_SUCCESS**
  - the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**
  - the library was not initialized.
- **CUSOLVER_STATUS_ALLOC_FAILED**
  - the resources could not be allocated.
- **CUSOLVER_STATUS_INVALID_VALUE**
  - invalid parameters were passed `m, n, nnz<=0`, base index is not 0 or 1.
- **CUSOLVER_STATUS_ARCH_MISMATCH**
  - the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**
  - an internal operation failed.
- **CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED**
  - the matrix type is not supported.

#### 2.5.2.5. cusolverSp<t>csreigvsi()

The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t cusolverSpScsreigvsi[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    float mu0,
    const float *x0,
    int maxite,
    float tol,
    float *mu,
    float *x);

cusolverStatus_t cusolverSpDcsreigvsi[Host](cusolverSpHandle_t handle,
    int m,
```

**cuSOLVER Library**
int nnz,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
double mu0,
const double *x0,
int maxite,
double tol,
double *mu,
double *x);

The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverSpCcsreigvsi[Host](cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const cuComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
cuComplex mu0,
const cuComplex *x0,
int maxite,
float tol,
cuComplex *mu,
cuComplex *x);

cusolverStatus_t
cusolverSpZcsreigvsi(cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
cuDoubleComplex mu0,
const cuDoubleComplex *x0,
int maxite,
double tol,
cuDoubleComplex *mu,
cuDoubleComplex *x);

This function solves the simple eigenvalue problem \( A^*x = \lambda^*x \) by shift-inverse method.

\( A \) is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays \( csrValA, \)
\( csrRowPtrA, \) and \( csrColIndA. \) The output parameter \( x \) is the approximated eigenvector of size \( m, \)

The following shift-inverse method corrects eigenpair step-by-step until convergence.

It accepts several parameters:

\( mu0 \) is an initial guess of eigenvalue. The shift-inverse method will converge to the eigenvalue \( \mu \) nearest \( \mu0 \) if \( \mu \) is a singleton. Otherwise, the shift-inverse method may not converge.

\( x0 \) is an initial eigenvector. If the user has no preference, just chose \( x0 \) randomly. \( x0 \) must be nonzero. It can be non-unit length.
tol is the tolerance to decide convergence. If tol is less than zero, it would be treated as zero.

maxite is maximum number of iterations. It is useful when shift-inverse method does not converge because the tolerance is too small or the desired eigenvalue is not a singleton.

**Shift-Inverse Method**

Given a initial guess of eigenvalue \( \mu_0 \) and initial vector \( x_0 \) of unit length

\[
\begin{align*}
\text{for } j = 0 : \text{maxite} \\
& x^{(0)} = x_0 \\
& \text{solve } A x^{(k+1)} = \mu_0 I - \mu x^{(k+1)} \\
& \text{compute approx. eigenvalue } \mu = x^H A x^{(k+1)} \\
& \text{normalize to unit length} \\
& \text{endfor}
\end{align*}
\]

The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. If \( A \) is symmetric/Hermitian and only lower/upper part is used or meaningful, the user has to extend the matrix into its missing upper/lower part, otherwise the result is wrong.

Remark 1: [cu|h]solver[S|D]csreigvsi only allows \( \mu_0 \) as a real number. This works if \( A \) is symmetric. Otherwise, the non-real eigenvalue has a conjugate counterpart on the complex plan, and shift-inverse method would not converge to such eigenvalue even the eigenvalue is a singleton. The user has to extend \( A \) to complex numbre and call [cu|h]solver[C|Z]csreigvsi with \( \mu_0 \) not on real axis.

Remark 2: the tolerance tol should not be smaller than \(|\mu_0|*\text{eps}\), where eps is machine zero. Otherwise, shift-inverse may not converge because of small tolerance.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolver library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix ( A ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of ( m + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

csrColIndA | device | host | integer array of nnz (= csrRowPtrA(m) - csrRowPtrA(0)) column indices of the nonzero elements of matrix A.
mu0 | host | host | initial guess of eigenvalue.
x0 | device | host | initial guess of eigenvector, a vector of size m.
maxite | host | host | maximum iterations in shift-inverse method.
tol | host | host | tolerance for convergence.

Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mu</td>
<td>device</td>
<td>host</td>
<td>approximated eigenvalue nearest mu0 under tolerance.</td>
</tr>
<tr>
<td>x</td>
<td>device</td>
<td>host</td>
<td>approximated eigenvector of size m.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [m, nnz&lt;=0], base index is not 0 or 1.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>

2.5.2.6. cusolverSp<t>csreigs()

cusolverStatus_t
cusolverSpScrreigs[Host](cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const float *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
cuComplex left_bottom_corner,
cuComplex right_upper_corner,
int *num_eigs);

cusolverStatus_t
cusolverSpDcsreigs[Host](cusolverSpHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
This function computes number of algebraic eigenvalues in a given box \( B \) by contour integral

\[
\text{number of algebraic eigenvalues in box } B = \frac{1}{2 \pi} \int_{C} \frac{P'(z)}{P(z)} \, dz
\]

where closed line \( C \) is boundary of the box \( B \) which is a rectangle specified by two points, one is left bottom corner (input parameter \texttt{left_bottom_corner}) and the other is right upper corner (input parameter \texttt{right_upper_corner}). \( P(z) = \det(A - zI) \) is the characteristic polynomial of \( A \).

\( A \) is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays \texttt{csrValA}, \texttt{csrRowPtrA}, and \texttt{csrColIndA}.

The output parameter \texttt{num_eigs} is number of algebraic eigenvalues in the box \( B \). This number may not be accurate due to several reasons:

1. the contour \( C \) is close to some eigenvalues or even passes through some eigenvalues.
2. the numerical integration is not accurate due to coarse grid size. The default resolution is 1200 grids along contour \( C \) uniformly.

Even though \texttt{csreigs} may not be accurate, it still can give the user some idea how many eigenvalues in a region where the resolution of disk theorem is bad. For example, standard 3-point stencil of finite difference of Laplacian operator is a tridiagonal matrix, and disk theorem would show “all eigenvalues are in the interval \([0, 4N^2]\)” where \( N \) is number of grids. In this case, \texttt{csreigs} is useful for any interval inside \([0, 4N^2]\).
Remark 1: if \( A \) is symmetric in real or hermitian in complex, all eigenvalues are real. The user still needs to specify a box, not an interval. The height of the box can be much smaller than the width.

Remark 2: only CPU (Host) path is provided.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>host</td>
<td>&lt;type&gt; array of ( \text{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>host</td>
<td>integer array of ( m + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>host</td>
<td>integer array of ( \text{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>left_bottom_corner</td>
<td>host</td>
<td>host</td>
<td>left bottom corner of the box.</td>
</tr>
<tr>
<td>right_upper_corner</td>
<td>host</td>
<td>host</td>
<td>right upper corner of the box.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_eigs</td>
<td>host</td>
<td>host</td>
<td>number of algebraic eigenvalues in a box.</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS** the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED** the library was not initialized.
- **CUSOLVER_STATUS_ALLOC_FAILED** the resources could not be allocated.
- **CUSOLVER_STATUS_INVALID_VALUE** invalid parameters were passed \([m, \text{nnz} <= 0], \) base index is not 0 or 1.
- **CUSOLVER_STATUS_ARCH_MISMATCH** the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR** an internal operation failed.
- **CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED** the matrix type is not supported.
2.5.3. Low Level Function Reference

This section describes low level API of cuSolverSP, including symrcm and batched QR.

2.5.3.1. cusolverSpXcsrsymrcm()

cusolverStatus_t
cusolverSpXcsrsymrcmHost(cusolverSpHandle_t handle, int n, int nnzA, const cusparseMatDescr_t descrA, const int *csrRowPtrA, const int *csrColIndA, int *p);

This function implements Symmetric Reverse Cuthill-McKee permutation. It returns a permutation vector \( p \) such that \( A(p,p) \) would concentrate nonzeros to diagonal. This is equivalent to \texttt{symrcm} in MATLAB, however the result may not be the same because of different heuristics in the pseudoperipheral finder. The \texttt{cuSolverSP} library implements \texttt{symrcm} based on the following two papers:


The output parameter \( p \) is an integer array of \( n \) elements. It represents a permutation array and it indexed using the base-0 convention. The permutation array \( p \) corresponds to a permutation matrix \( P \), and satisfies the following relation:

\[
A(p,p) = P^T A^T P
\]

\( A \) is an \( n \times n \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \).

The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}. Internally \texttt{rcm} works on \( A + A^T \), the user does not need to extend the matrix if the matrix is not symmetric.

Remark 1: only CPU (Host) path is provided.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ). It is the size of ( \text{csrValA} ) and ( \text{csrColIndA} ).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

2.5.3.2. cusolverSpXcsrsymmdq()

cusolverStatus_t
cusolverSpXcsrsymmdqHost(cusolverSpHandle_t handle,
  int n,
  int nnzA,
  const cusparseMatDescr_t descrA,
  const int *csrRowPtrA,
  const int *csrColIndA,
  int *p);

This function implements Symmetric Minimum Degree Algorithm based on Quotient Graph. It returns a permutation vector p such that A(p, p) would have less zero fill-in during Cholesky factorization. The cuSolverSP library implements symmdq based on the following two papers:


The output parameter p is an integer array of n elements. It represents a permutation array with base-0 index. The permutation array p corresponds to a permutation matrix P, and satisfies the following relation:
A is an n×n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Internally mdq works on \( A + A^T \), the user does not need to extend the matrix if the matrix is not symmetric.

Remark 1: only CPU [Host] path is provided.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ). It is the size of csrValA and csrColIndA.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>host</td>
<td>integer array of ( n+1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>host</td>
<td>integer array of ( nnzA ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>host</td>
</tr>
</tbody>
</table>

Status Returned

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_NOT_INITIALIZED: the library was not initialized.
- CUSOLVER_STATUS_ALLOC_FAILED: the resources could not be allocated.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed \( [n, nnzA] \leq 0 \), base index is not 0 or 1.
- CUSOLVER_STATUS_ARCH_MISMATCH: the device only supports compute capability 2.0 and above.
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.
- CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED: the matrix type is not supported.

### 2.5.3.3. cusolverSpXcsrsymamd()

cusolverStatus_t

cusolverSpXcsrsymamdHost(cusolverSpHandle_t handle,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const int *csrRowPtrA,
const int *csrColIndA,
int *p);

This function implements Symmetric Approximate Minimum Degree Algorithm based on Quotient Graph. It returns a permutation vector $p$ such that $A(p,p)$ would have less zero fill-in during Cholesky factorization. The cuSolverSP library implements symamd based on the following paper:


The output parameter $p$ is an integer array of $n$ elements. It represents a permutation array with base-0 index. The permutation array $p$ corresponds to a permutation matrix $P$, and satisfies the following relation:

$$A(p,p) = P^T A P$$

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

The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Internally amd works on $A + A^T$, the user does not need to extend the matrix if the matrix is not symmetric.

Remark 1: only CPU (Host) path is provided.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix $A$. It is the size of $csrValA$ and $csrColIndA$.</td>
</tr>
</tbody>
</table>
| descrA        | host           | the descriptor of matrix $A$. The supported matrix type is
|               |                | CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are
|               |                | CUSPARSE_INDEX_BASE_ZERO and                                  |
|               |                | CUSPARSE_INDEX_BASE_ONE.                                                  |
| csrRowPtrA    | host           | integer array of $n+1$ elements that contains the start of every row and
|               |                | the end of the last row plus one.                                       |
| csrColIndA    | host           | integer array of $nnzA$ column indices of the nonzero elements of
|               |                | matrix $A$.                                                              |

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsolver</td>
<td>permutation vector of size $n$.</td>
</tr>
</tbody>
</table>

**Status Returned**
2.5.3.4. cusolverSpXcsrmetisnd()

This function is a wrapper of METIS_NodeND. It returns a permutation vector \( p \) such that \( A(p,p) \) would have less zero fill-in during nested dissection. The cuSolverSP library links libmetis_static.a which is 64-bit metis-5.1.0.

The parameter \( \text{options} \) is the configuration of \texttt{metis}. For those who do not have experiences of \texttt{metis}, set \( \text{options} = \text{NULL} \) for default setting.

The output parameter \( p \) is an integer array of \( n \) elements. It represents a permutation array with base-0 index. The permutation array \( p \) corresponds to a permutation matrix \( P \), and satisfies the following relation:

\[
A(p,p) = P^* A^* P^T
\]

\( A \) is an \( n \times n \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \).

The supported matrix type is \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}. Internally \texttt{csrmetisnd} works on \( A + A^T \), the user does not need to extend the matrix if the matrix is not symmetric.

Remark 1: only CPU (Host) path is provided.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>( n )</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{nnzA} )</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ). It is the size of ( \text{csrValA} ) and ( \text{csrColIndA} ).</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

```
| descrA    | host       | the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE. |
| csrRowPtrA | host       | integer array of n+1 elements that contains the start of every row and the end of the last row plus one. |
| csrColIndA | host       | integer array of nnzA column indices of the nonzero elements of matrix A. |
| options   | host       | integer array to configure metis. |

Output

```

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>host</td>
<td>permutation vector of size n.</td>
</tr>
</tbody>
</table>

Status Returned

```

| CUSOLVER_STATUS_SUCCESS          | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED  | the library was not initialized. |
| CUSOLVER_STATUS_ALLOC_FAILED     | the resources could not be allocated. |
| CUSOLVER_STATUS_INVALID_VALUE    | invalid parameters were passed \( n, \text{nnzA} \leq 0 \), base index is not 0 or 1. |
| CUSOLVER_STATUS_INTERNAL_ERROR   | an internal operation failed. |
| CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED | the matrix type is not supported. |

2.5.3.5. cusolverSpXcsrzfd()```

```
cusolverStatus_t
cusolverSpXcsrzfdHost(
    cusolverSpHandle_t handle,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int *P,
    int *numnz)

cusolverStatus_t
cusolverSpDcsrzfdHost(
    cusolverSpHandle_t handle,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int *P,
    int *numnz)
```

```
This function implements MC21, zero-free diagonal algorithm. It returns a permutation vector \( p \) such that \( A(p,:) \) has no zero diagonal.

\( A \) is an \( n \times n \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \). The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}.

The output parameter \( p \) is an integer array of \( n \) elements. It represents a permutation array with base-0 index. The permutation array \( p \) corresponds to a permutation matrix \( P \), and satisfies the following relation:

\[
A(p,:) = P^T A
\]

The output parameter \( \text{numnz} \) describes number of nonzero diagonal in permuted matrix \( A(p,:) \). If \( \text{numnz} \) is less than \( n \), matrix \( A \) has structural singularity.

Remark 1: only CPU (Host) path is provided.

Remark 2: this routine does not maximize diagonal value of permuted matrix. The user cannot expect this routine can make “LU without pivoting” stable.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>*Host MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ). It is the size of ( \text{csrValA} ) and ( \text{csrColIndA} ).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are</td>
</tr>
</tbody>
</table>
## Using the CUSOLVER API

### cuSOLVER Library

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValA</td>
<td>host</td>
<td>&lt;type&gt; array of nnzA ( = csrRowPtrA(m) - csrRowPtrA(0)) nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>host</td>
<td>integer array of n+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>host</td>
<td>integer array of nnzA column indices of the nonzero elements of matrix A.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>*Host MemSpace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>host</td>
<td>permutation vector of size n.</td>
</tr>
<tr>
<td>numnz</td>
<td>host</td>
<td>number of nonzeros on diagonal of permuted matrix.</td>
</tr>
</tbody>
</table>

### Status Returned

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_ALLOC_FAILED**: the resources could not be allocated.
- **CUSOLVER_STATUS_INVALID_VALUE**: invalid parameters were passed \( n, nnzA <= 0 \), base index is not 0 or 1.
- **CUSOLVER_STATUS_ARCH_MISMATCH**: the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.
- **CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED**: the matrix type is not supported.

### 2.5.3.6. cusolverSpXcsrperm()

```c
 cusolverStatus_t cusolverSpXcsrperm_bufferSizeHost(cusolverSpHandle_t handle, 
          int m, 
          int n, 
          int nnzA, 
          const cusparseMatDescr_t descrA, 
          int *csrRowPtrA, 
          int *csrColIndA, 
          const int *p, 
          const int *q, 
          size_t *bufferSizeInBytes);
```

```c
 cusolverStatus_t cusolverSpXcsrpermHost(cusolverSpHandle_t handle, 
          int m, 
          int n, 
          int nnzA, 
          const cusparseMatDescr_t descrA, 
          int *csrRowPtrA, 
          int *csrColIndA, 
```

### cuSOLVER Library

DU-06709-001_v11.1 | 238
Given a left permutation vector \( p \) which corresponds to permutation matrix \( P \) and a right permutation vector \( q \) which corresponds to permutation matrix \( Q \), this function computes permutation of matrix \( A \) by

\[
B = P \cdot A \cdot Q^T
\]

\( A \) is an \( m \times n \) sparse matrix that is defined in CSR storage format by the three arrays  \( \text{csrValA} \), \( \text{csrRowPtrA} \) and \( \text{csrColIndA} \).

The operation is in-place, i.e. the matrix \( A \) is overwritten by \( B \).

The permutation vector \( p \) and \( q \) are base 0. \( p \) performs row permutation while \( q \) performs column permutation. One can also use MATLAB command \( B = A(p,q) \) to permute matrix \( A \).

This function only computes sparsity pattern of \( B \). The user can use parameter \( \text{map} \) to get \( \text{csrValB} \) as well. The parameter \( \text{map} \) is an input/output. If the user sets \( \text{map}=0:1:(\text{nnzA}-1) \) before calling \( \text{csrperm} \), \( \text{csrValB}=\text{csrValA}(\text{map}) \).

The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. If \( A \) is symmetric and only lower/upper part is provided, the user has to pass \( A + A^T \) into this function.

This function requires a buffer size returned by \( \text{csrperm_bufferSize()} \). The address of \( \text{pBuffer} \) must be a multiple of 128 bytes. If it is not, \text{CUSOLVER_STATUS_INVALID_VALUE} is returned.

For example, if matrix \( A \) is

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
7.0 & 8.0 & 9.0
\end{bmatrix}
\]

and left permutation vector \( p=(0, 2, 1) \), right permutation vector \( q=(2, 1, 0) \), then \( P \cdot A \cdot Q^T \) is

\[
P \cdot A \cdot Q^T = \begin{bmatrix}
3.0 & 2.0 & 1.0 \\
9.0 & 8.0 & 7.0 \\
6.0 & 5.0 & 4.0
\end{bmatrix}
\]

Remark 1: only CPU (Host) path is provided.

Remark 2: the user can combine \( \text{csrsymrcm} \) and \( \text{csrperm} \) to get \( P \cdot A \cdot \text{P}^T \) which has less zero fill-in during QR factorization.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSpMemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolver library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix ( A ).</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix $A$. It is the size of $csrValA$ and $csrColIndA$.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>host</td>
<td>integer array of $m+1$ elements that contains the start of every row and end of last row plus one of matrix $A$.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>host</td>
<td>integer array of $nnzA$ column indices of the nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>p</td>
<td>host</td>
<td>left permutation vector of size $m$.</td>
</tr>
<tr>
<td>q</td>
<td>host</td>
<td>right permutation vector of size $n$.</td>
</tr>
<tr>
<td>map</td>
<td>host</td>
<td>integer array of $nnzA$ indices. If the user wants to get relationship between $A$ and $B$, map must be set $0:1:(nnzA-1)$.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>host</td>
<td>buffer allocated by the user, the size is returned by $csrperm_bufferSize()$.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrRowPtrA</td>
<td>host</td>
<td>integer array of $m+1$ elements that contains the start of every row and end of last row plus one of matrix $B$.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>host</td>
<td>integer array of $nnzA$ column indices of the nonzero elements of matrix $B$.</td>
</tr>
<tr>
<td>map</td>
<td>host</td>
<td>integer array of $nnzA$ indices that maps matrix $A$ to matrix $B$.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed $[m, n, nnzA]=0$, base index is not 0 or 1.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>
2.5.3.7. **cusolverSpXcsrqrBatched()**

The create and destroy methods start and end the lifetime of a csrqrInfo object.

```c
cusolverStatus_t
cusolverSpCreateCsrqrInfo(csrqrInfo_t *info);

cusolverStatus_t
cusolverSpDestroyCsrqrInfo(csrqrInfo_t info);
```

Analysis is the same for all data types, but each data type has a unique buffer size.

```c
cusolverStatus_t
cusolverSpXcsrqrAnalysisBatched(cusolverSpHandle_t handle,  
  int m,  
  int n,  
  int nnzA,  
  const cusparseMatDescr_t descrA,  
  const int *csrRowPtrA,  
  const int *csrColIndA,  
  csrqrInfo_t info);
```

```c
cusolverStatus_t
cusolverSpScsrqrBufferInfoBatched(cusolverSpHandle_t handle,  
  int m,  
  int n,  
  int nnzA,  
  const cusparseMatDescr_t descrA,  
  const float *csrValA,  
  const int *csrRowPtrA,  
  const int *csrColIndA,  
  int batchSize,  
  csrqrInfo_t info,  
  size_t *internalDataInBytes,  
  size_t *workspaceInBytes);
```

```c
cusolverStatus_t
cusolverSpDcsrqrBufferInfoBatched(cusolverSpHandle_t handle,  
  int m,  
  int n,  
  int nnzA,  
  const cusparseMatDescr_t descrA,  
  const double *csrValA,  
  const int *csrRowPtrA,  
  const int *csrColIndA,  
  int batchSize,  
  csrqrInfo_t info,  
  size_t *internalDataInBytes,  
  size_t *workspaceInBytes);
```

```c
cusolverStatus_t
cusolverSpCcsrqrBufferInfoBatched(cusolverSpHandle_t handle,  
  int m,  
  int n,  
  int nnzA,  
  const cusparseMatDescr_t descrA,  
  const cuComplex *csrValA,
  const cuComplex *csrRowPtrA,
  const cuComplex *csrColIndA,
  int batchSize,
  csrqrInfo_t info,
  size_t *internalDataInBytes,
  size_t *workspaceInBytes);
```

Calculate buffer sizes for complex valued data types.
Using the CUSOLVER API

```c
const int *csrRowPtrA,
const int *csrColIndA,
int batchSize,
csrqrInfo_t info,
size_t *internalDataInBytes,
size_t *workspaceInBytes);

cusolverStatus_t
cusolverSpZcsrqrBufferInfoBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
int batchSize,
csrqrInfo_t info,
size_t *internalDataInBytes,
size_t *workspaceInBytes);

The S and D data types are real valued single and double precision, respectively.

```c
const float *csrValA,
const float *b,
float *x,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

cusolverStatus_t
cusolverSpScsrqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const float *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const float *b,
float *x,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

cusolverStatus_t
cusolverSpDcsrqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const double *b,
double *x,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

The C and Z data types are complex valued single and double precision, respectively.

```c
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

cusolverStatus_t
cusolverSpCcsrqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

cusolverStatus_t
cusolverSpDcsrqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const double *b,
double *x,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

The C and Z data types are complex valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverSpCcspqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const cuDoubleComplex *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

cusolverStatus_t
cusolverSpDcspqrsvBatched(cusolverSpHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const double *b,
double *x,
int batchSize,
csrqrInfo_t info,
void *pBuffer);

```
The batched sparse QR factorization is used to solve either a set of least-squares problems

\[ x_j = \text{argmin} \left| A_j^* z - b_j \right|, \quad j = 1, 2, ..., \text{batchSize} \]

or a set of linear systems

\[ A_j x_j = b_j, \quad j = 1, 2, ..., \text{batchSize} \]

where each \( A_j \) is a \( m \times n \) sparse matrix that is defined in CSR storage format by the four arrays \( \text{csrValA}, \text{csrRowPtrA}, \text{csrColIndA} \).

The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. If \( A \) is symmetric and only lower/upper part is provided, the user has to pass \( A + A^H \) into this function.

The prerequisite to use batched sparse QR has two-folds. First all matrices \( A_j \) must have the same sparsity pattern. Second, no column pivoting is used in least-square problem, so the solution is valid only if \( A_j \) is of full rank for all \( j = 1, 2, ..., \text{batchSize} \). All matrices have the same sparsity pattern, so only one copy of \( \text{csrRowPtrA} \) and \( \text{csrColIndA} \) is used. But the array \( \text{csrValA} \) stores coefficients of \( A_j \) one after another. In other words, \( \text{csrValA}[k \times \text{nnzA} : (k+1) \times \text{nnzA}] \) is the value of \( A_k \).

The batched QR uses opaque data structure \( \text{csrqrInfo} \) to keep intermediate data, for example, matrix \( Q \) and matrix \( R \) of QR factorization. The user needs to create \( \text{csrqrInfo} \) first by \text{cusolverSpCreateCsrqrInfo} before any function in batched QR operation. The \( \text{csrqrInfo} \) would not release internal data until \text{cusolverSpDestroyCsrqrInfo} is called.

There are three routines in batched sparse QR, \text{cusolverSpXcsrqrAnalysisBatched}, \text{cusolverSp[S|D|C|Z]csrqrBufferInfoBatched} and \text{cusolverSp[S|D|C|Z]csrqrsvBatched}.

First, \text{cusolverSpXcsrqrAnalysisBatched} is the analysis phase, used to analyze sparsity pattern of matrix \( Q \) and matrix \( R \) of QR factorization. Also parallelism is extracted during analysis phase. Once analysis phase is done, the size of working space to perform QR is
Using the CUSOLVER API

known. However, `cusolverSpXcsrqrAnalysisBatched` uses CPU to analyze the structure of matrix `A`, and this may consume a lot of memory. If host memory is not sufficient to finish the analysis, `CUSOLVER_STATUS_ALLOC_FAILED` is returned. The required memory for analysis is proportional to zero fill-in in QR factorization. The user may need to perform some kind of reordering to minimize zero fill-in, for example, `colamd` or `symrcm` in MATLAB. `cusolverSp` library provides `symrcm` (`cusolverSpXcsrsymrcm`).

Second, the user needs to choose proper `batchSize` and to prepare working space for sparse QR. There are two memory blocks used in batched sparse QR. One is internal memory block used to store matrix `Q` and matrix `R`. The other is working space used to perform numerical factorization. The size of the former is proportional to `batchSize`, and the size is specified by returned parameter `internalDataInBytes` of `cusolverSp[S|D|C|Z]csrqrBufferInfoBatched`. While the size of the latter is almost independent of `batchSize`, and the size is specified by returned parameter `workspaceInBytes` of `cusolverSp[S|D|C|Z]csrqrBufferInfoBatched`. The internal memory block is allocated implicitly during first call of `cusolverSp[S|D|C|Z]csrqrsvBatched`. The user only needs to allocate working space for `cusolverSp[S|D|C|Z]csrqrsvBatched`.

Instead of trying all batched matrices, the user can find maximum `batchSize` by querying `cusolverSp[S|D|C|Z]csrqrBufferInfoBatched`. For example, the user can increase `batchSize` till summation of `internalDataInBytes` and `workspaceInBytes` is greater than size of available device memory.

Suppose that the user needs to perform 253 linear solvers and available device memory is 2GB. If `cusolverSp[S|D|C|Z]csrqrsvBatched` can only afford `batchSize` 100, the user has to call `cusolverSp[S|D|C|Z]csrqrsvBatched` three times to finish all. The user calls `cusolverSp[S|D|C|Z]csrqrBufferInfoBatched` with `batchSize` 100. The opaque info would remember this `batchSize` and any subsequent call of `cusolverSp[S|D|C|Z]csrqrsvBatched` cannot exceed this value. In this example, the first two calls of `cusolverSp[S|D|C|Z]csrqrsvBatched` will use `batchSize` 100, and last call of `cusolverSp[S|D|C|Z]csrqrsvBatched` will use `batchSize` 53.

Example: suppose that `A0`, `A1`, ..., `A9` have the same sparsity pattern, the following code solves 10 linear systems \( A_j x_j = b_j \), \( j = 0, 2, \ldots, 9 \) by batched sparse QR.

```c
// Suppose that A0, A1, ..., A9 are m x m sparse matrix represented by CSR format,
// Each matrix Aj has nonzero nnzA, and shares the same csrRowPtrA and csrColIndA.
// csrValA is aggregation of A0, A1, ..., A9.
int m ; // number of rows and columns of each Aj
int nnzA ; // number of nonzeros of each Aj
int *csrRowPtrA ; // each Aj has the same csrRowPtrA
int *csrColIndA ; // each Aj has the same csrColIndA
double *csrValA ; // aggregation of A0,A1,...,A9
cont int batchSize = 10; // 10 linear systems

cusolverSpHandle_t handle; // handle to cusolver library
csrqrInfo_t info = NULL;
cusparseMatDescr_t descrA = NULL;
void *pBuffer = NULL; // working space for numerical factorization

// step 1: create a descriptor
cusparseCreateMatDescr(&descrA);
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE); // A is base-1
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL); // A is a general matrix

// step 2: create empty info structure
cusolverSpCreateCsrqrInfo(&info);
```
Using the CUSOLVER API

// step 3: symbolic analysis
cusolverSpXcsrqrAnalysisBatched(
    handle, m, m, nnzA,
    descrA, csrRowPtrA, csrColIndA, info);

// step 4: allocate working space for Aj*xj=bj
cusolverSpDcsrqrBufferInfoBatched(
    handle, m, m, nnzA,
    descrA, csrValA, csrRowPtrA, csrColIndA,
    batchSize, info,
    &internalDataInBytes,
    &workspaceInBytes);
cudaMalloc(&pBuffer, workspaceInBytes);

// step 5: solve Aj*xj = bj
cusolverSpDcsrqrsvBatched(
    handle, m, m, nnzA,
    descrA, csrValA, csrRowPtrA, csrColIndA, b,
    x, batchSize, info,
    pBuffer);

// step 7: destroy info
cusolverSpDestroyCsrqrInfo(info);

Please refer to Appendix B for detailed examples.

Remark 1: only GPU [device] path is provided.

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>cusolverSp MemSpace</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of each matrix Aj.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of each matrix Aj.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of each matrix Aj. It is the size csrColIndA.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of each matrix Aj. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>&lt;type&gt; array of nnzA*batchSize nonzero elements of matrices A0, A1, .... All matrices are aggregated one after another.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>integer array of nnzA column indices of the nonzero elements of each matrix Aj.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

cuSOLVER Library

DU-06709-001_v11.1   |   246

2.6. cuSolverRF: Refactorization Reference

This chapter describes API of cuSolverRF, a library for fast refactorization.

2.6.1. cusolverRfAccessBundledFactors()

```c
 cusolverStatus_t
 cusolverRfAccessBundledFactors(/* Input */
     cusolverRfHandle_t handle,
     /* Output (in the host memory) */
     int* nnzM,
     /* Output (in the device memory) */
     int** Mp,
     int** Mi,
```
double** Mx);  

This routine allows direct access to the lower \( L \) and upper \( U \) triangular factors stored in the cuSolverRF library handle. The factors are compressed into a single matrix \( M = (L - I) + U \), where the unitary diagonal of \( L \) is not stored. It is assumed that a prior call to the `cusolverRfRefactor()` was done in order to generate these triangular factors.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>nnzM</td>
<td>host</td>
<td>output</td>
<td>the number of non-zero elements of matrix ( M ).</td>
</tr>
<tr>
<td>Mp</td>
<td>device</td>
<td>output</td>
<td>the array of offsets corresponding to the start of each row in the arrays ( M_1 ) and ( M_x ). This array has also an extra entry at the end that stores the number of non-zero elements in the matrix ( M ). The array size is ( n+1 ).</td>
</tr>
<tr>
<td>Mi</td>
<td>device</td>
<td>output</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix ( M ). It is assumed that this array is sorted by row and by column within each row. The array size is ( nnzM ).</td>
</tr>
<tr>
<td>Mx</td>
<td>device</td>
<td>output</td>
<td>the array of values corresponding to the non-zero elements in the matrix ( M ). It is assumed that this array is sorted by row and by column within each row. The array size is ( nnzM ).</td>
</tr>
</tbody>
</table>

**Status Returned**

- `CUSOLVER_STATUS_SUCCESS` the operation completed successfully.
- `CUSOLVER_STATUS_NOT_INITIALIZED` the library was not initialized.
- `CUSOLVER_STATUS_EXECUTION_FAILED` a kernel failed to launch on the GPU.

### 2.6.2. `cusolverRfAnalyze()`

cusolverStatus_t cusolverRfAnalyze(cusolverRfHandle_t handle);

This routine performs the appropriate analysis of parallelism available in the LU re-factorization depending upon the algorithm chosen by the user.

\[ A = L \times U \]

It is assumed that a prior call to the `cusolverRfSetup[Host|Device]()` was done in order to create internal data structures needed for the analysis.

This routine needs to be called only once for a single linear system

\[ A_ix = f_i \]
2.6.3. cusolverRfSetupDevice()

```c
cusolverStatus_t
cusolverRfSetupDevice(/* Input (in the device memory) */
    int n,
    int nnzA,
    int* csrRowPtrA,
    int* csrColIndA,
    double* csrValA,
    int nnzL,
    int* csrRowPtrL,
    int* csrColIndL,
    double* csrValL,
    int nnzU,
    int* csrRowPtrU,
    int* csrColIndU,
    double* csrValU,
    int* P,
    int* Q,
    /* Output */
    cusolverRfHandle_t handle);
```

This routine assembles the internal data structures of the cuSolverRF library. It is often the first routine to be called after the call to the `cusolverRfCreate()` routine.

This routine accepts as input (on the device) the original matrix $A$, the lower ($L$) and upper ($U$) triangular factors, as well as the left ($P$) and the right ($Q$) permutations resulting from the full LU factorization of the first ($i=1$) linear system

$$A_1x_i = f_i$$

The permutations $P$ and $Q$ represent the final composition of all the left and right reorderings applied to the original matrix $A$, respectively. However, these permutations are often associated with partial pivoting and reordering to minimize fill-in, respectively.

This routine needs to be called only once for a single linear system

$$A_i x_i = f_i$$

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>in/out</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.
- **CUSOLVER_STATUS_EXECUTION_FAILED**: a kernel failed to launch on the GPU.
- **CUSOLVER_STATUS_ALLOC_FAILED**: an allocation of memory failed.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix A.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays csrColIndA and csrValA. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix. The array size is n+1.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is nnzA.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is nnzA.</td>
</tr>
<tr>
<td>nnzL</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix L.</td>
</tr>
<tr>
<td>csrRowPtrL</td>
<td>device</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays csrColIndL and csrValL. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix L. The array size is n+1.</td>
</tr>
<tr>
<td>csrColIndL</td>
<td>device</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix L. It is assumed that this array is sorted by row and by column within each row. The array size is nnzL.</td>
</tr>
<tr>
<td>csrValL</td>
<td>device</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the matrix L. It is assumed that this array is sorted by row and by column within each row. The array size is nnzL.</td>
</tr>
<tr>
<td>nnzU</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix U.</td>
</tr>
<tr>
<td>csrRowPtrU</td>
<td>device</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays csrColIndU and csrValU. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix U. The array size is n+1.</td>
</tr>
</tbody>
</table>
| csrColIndU| device   | input        | the array of column indices corresponding to the non-zero elements in the matrix U. It is assumed that this
2.6.4. cusolverRfSetupHost()

cusolverStatus_t

cusolverRfSetupHost(/* Input (in the host memory) */
    int n,
    int nnzA,
    int* h_csrRowPtrA,
    int* h_csrColIndA,
    double* h_csrValA,
    int nnzL,
    int* h_csrRowPtrL,
    int* h_csrColIndL,
    double* h_csrValL,
    int nnzU,
    int* h_csrRowPtrU,
    int* h_csrColIndU,
    double* h_csrValU,
    int* h_P,
    int* h_Q,
    /* Output */
    cusolverRfHandle_t handle);

This routine assembles the internal data structures of the cuSolverRF library. It is often the first routine to be called after the call to the cusolverRfCreate() routine.

This routine accepts as input (on the host) the original matrix A, the lower (L) and upper (U) triangular factors, as well as the left (P) and the right (Q) permutations resulting from the full LU factorization of the first (i=1) linear system.
Using the CUSOLVER API

\[ A_i x_i = f_i \]

The permutations \( p \) and \( q \) represent the final composition of all the left and right reorderings applied to the original matrix \( A \), respectively. However, these permutations are often associated with partial pivoting and reordering to minimize fill-in, respectively.

This routine needs to be called only once for a single linear system

\[ A_i x_i = f_i \]

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix ( A ).</td>
</tr>
<tr>
<td>h_csrRowPtrA</td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>arrays ( h_{csrColIndA} ) and ( h_{csrValA} ). This array has also</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>an extra entry at the end that stores the number of non-zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elements in the matrix. The array size is ( n+1 ).</td>
</tr>
<tr>
<td>h_csrColIndA</td>
<td>host</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in the matrix. It is assumed that this array is sorted by row and by</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>column within each row. The array size is ( nnzA ).</td>
</tr>
<tr>
<td>h_csrValA</td>
<td>host</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>matrix. It is assumed that this array is sorted by row and by column</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>within each row. The array size is ( nnzA ).</td>
</tr>
<tr>
<td>nnzL</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix ( L ).</td>
</tr>
<tr>
<td>h_csrRowPtrL</td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>arrays ( h_{csrColIndL} ) and ( h_{csrValL} ). This array has also</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>an extra entry at the end that stores the number of non-zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>elements in the matrix ( L ). The array size is ( n+1 ).</td>
</tr>
<tr>
<td>h_csrColIndL</td>
<td>host</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in the matrix ( L ). It is assumed that this array is sorted by row</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>and by column within each row. The array size is ( nnzL ).</td>
</tr>
<tr>
<td>h_csrValL</td>
<td>host</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>matrix ( L ). It is assumed that this array is sorted by row</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzU</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix $U$.</td>
</tr>
<tr>
<td>$h_{\text{csrRowPtrU}}$</td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays $h_{\text{csrColIndU}}$ and $h_{\text{csrValU}}$. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix $U$. The array size is $n+1$.</td>
</tr>
<tr>
<td>$h_{\text{csrColIndU}}$</td>
<td>host</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix $U$. It is assumed that this array is sorted by row and by column within each row. The array size is $nnzU$.</td>
</tr>
<tr>
<td>$h_{\text{csrValU}}$</td>
<td>host</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the matrix $U$. It is assumed that this array is sorted by row and by column within each row. The array size is $nnzU$.</td>
</tr>
<tr>
<td>$h_{P}$</td>
<td>host</td>
<td>input</td>
<td>the left permutation (often associated with pivoting). The array size in $n$.</td>
</tr>
<tr>
<td>$h_{Q}$</td>
<td>host</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in $n$.</td>
</tr>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>an unsupported value or parameter was passed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>an allocation of memory failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 2.6.5. cusolverRfCreate()

cusolverStatus_t cusolverRfCreate(cusolverRfHandle_t *handle);

This routine initializes the cuSolverRF library. It allocates required resources and must be called prior to any other cuSolverRF library routine.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the pointer to the cuSolverRF library handle.</td>
</tr>
</tbody>
</table>

**Status Returned**
CUSOLVER_STATUS_SUCCESS | the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED | the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED | an allocation of memory failed.
CUSOLVER_STATUS_INTERNAL_ERROR | an internal operation failed.

2.6.6. cusolverRfExtractBundledFactorsHost()

cusolverStatus_t

cusolverRfExtractBundledFactorsHost(/* Input */
    cusolverRfHandle_t handle,
    /* Output (in the host memory) */
    int* h_nnzM,
    int** h_Mp,
    int** h_Mi,
    double** h_Mx);

This routine extracts lower (L) and upper (U) triangular factors from the cuSolverRF library handle into the host memory. The factors are compressed into a single matrix $M=(L-I)+U$, where the unitary diagonal of (L) is not stored. It is assumed that a prior call to the cusolverRfRefactor() was done in order to generate these triangular factors.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>h_nnzM</td>
<td>host</td>
<td>output</td>
<td>the number of non-zero elements of matrix $M$.</td>
</tr>
<tr>
<td>h_Mp</td>
<td>host</td>
<td>output</td>
<td>the array of offsets corresponding to the start of each row in the arrays h_Mi and h_Mx. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix $M$. The array size is $n+1$.</td>
</tr>
<tr>
<td>h_Mi</td>
<td>host</td>
<td>output</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is h_nnzM.</td>
</tr>
<tr>
<td>h_Mx</td>
<td>host</td>
<td>output</td>
<td>the array of values corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is h_nnzM.</td>
</tr>
</tbody>
</table>

Status Returned

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED | the library was not initialized. |
| CUSOLVER_STATUS_ALLOC_FAILED | an allocation of memory failed. |
CUSOLVER_STATUS_EXECUTION_FAILED a kernel failed to launch on the GPU.

2.6.7. cusolverRfExtractSplitFactorsHost()

cusolverStatus_t
cusolverRfExtractSplitFactorsHost(/* Input */
cusolverRfHandle_t handle,
/* Output (in the host memory) */
int* h_nnzL,
int** h_Lp,
int** h_Li,
double** h_Lx,
int* h_nnzU,
int** h_Up,
int** h.Ui,
double** h_Ux);

This routine extracts lower ($L$) and upper ($U$) triangular factors from the cuSolverRF library handle into the host memory. It is assumed that a prior call to the cusolverRfRefactor() was done in order to generate these triangular factors.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>h_nnzL</td>
<td>host</td>
<td>output</td>
<td>the number of non-zero elements of matrix $L$.</td>
</tr>
<tr>
<td>h_Lp</td>
<td>host</td>
<td>output</td>
<td>the array of offsets corresponding to the start of each row in the arrays $h.Li$ and $h.Lx$. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix $L$. The array size is $n+1$.</td>
</tr>
<tr>
<td>h_Li</td>
<td>host</td>
<td>output</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix $L$. It is assumed that this array is sorted by row and by column within each row. The array size is $h.nnzL$.</td>
</tr>
<tr>
<td>h_Lx</td>
<td>host</td>
<td>output</td>
<td>the array of values corresponding to the non-zero elements in the matrix $L$. It is assumed that this array is sorted by row and by column within each row. The array size is $h.nnzL$.</td>
</tr>
<tr>
<td>h_nnzU</td>
<td>host</td>
<td>output</td>
<td>the number of non-zero elements of matrix $U$.</td>
</tr>
<tr>
<td>h_Up</td>
<td>host</td>
<td>output</td>
<td>the array of offsets corresponding to the start of each row in the arrays $h.Ui$ and $h.Ux$. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix $U$. The array size is $n+1$.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

The cuSOLVER Library

DU-06709-001_v11.1   |   255

h_Ui | host | output | the array of column indices corresponding to the non-zero elements in the matrix U. It is assumed that this array is sorted by row and by column within each row. The array size is $h_{nnz}\mu$.

h_Ux | host | output | the array of values corresponding to the non-zero elements in the matrix U. It is assumed that this array is sorted by row and by column within each row. The array size is $h_{nnz}\mu$.

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>an allocation of memory failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
</tbody>
</table>

2.6.8. cusolverRfDestroy()

cusolverStatus_t cusolverRfDestroy(cusolverRfHandle_t handle);

This routine shuts down the cuSolverRF library. It releases acquired resources and must be called after all the cuSolverRF library routines.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the cuSolverRF library handle.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

2.6.9. cusolverRfGetMatrixFormat()

cusolverStatus_t cusolverRfGetMatrixFormat(cusolverRfHandle_t handle,
                                            cusolverRfMatrixFormat_t *format,
                                            cusolverRfUnitDiagonal_t *diag);

This routine gets the matrix format used in the cusolverRfSetupDevice(), cusolverRfSetupHost(), cusolverRfResetValues(), cusolverRfExtractBundledFactorsHost() and cusolverRfExtractSplitFactorsHost() routines.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>
format host output the enumerated matrix format type.

diag host output the enumerated unit diagonal type.

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

### 2.6.10. cusolverRfGetNumericProperties()

```c
 cusolverStatus_t
cusolverRfGetNumericProperties(cusolverRfHandle_t handle,
                               double *zero,
                               double *boost);
```

This routine gets the numeric values used for checking for “zero” pivot and for boosting it in the cusolverRfRefactor() and cusolverRfSolve() routines. The numeric boosting will be used only if boost > 0.0.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>zero</td>
<td>host</td>
<td>output</td>
<td>the value below which zero pivot is flagged.</td>
</tr>
<tr>
<td>boost</td>
<td>host</td>
<td>output</td>
<td>the value which is substituted for zero pivot (if the later is flagged).</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

### 2.6.11. cusolverRfGetNumericBoostReport()

```c
 cusolverStatus_t
cusolverRfGetNumericBoostReport(cusolverRfHandle_t handle,
                                 cusolverRfNumericBoostReport_t *report);
```

This routine gets the report whether numeric boosting was used in the cusolverRfRefactor() and cusolverRfSolve() routines.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>report</td>
<td>host</td>
<td>output</td>
<td>the enumerated boosting report type.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>
2.6.12. cusolverRfGetResetValuesFastMode()

```c
 cusolverStatus_t cusolverRfGetResetValuesFastMode(cusolverRfHandle_t handle,
              rfResetValuesFastMode_t *fastMode);
```

This routine gets the mode used in the `cusolverRfResetValues` routine.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>fastMode</td>
<td>host</td>
<td>output</td>
<td>the enumerated mode type.</td>
</tr>
</tbody>
</table>

Status Returned

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.

2.6.13. cusolverRfGet_Algs()

```c
 cusolverStatus_t cusolverRfGet_Algs(cusolverRfHandle_t handle,
                                       cusolverRfFactorization_t* fact_alg,
                                       cusolverRfTriangularSolve_t* solve_alg);
```

This routine gets the algorithm used for the refactorization in `cusolverRfRefactor()` and the triangular solve in `cusolverRfSolve()`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>alg</td>
<td>host</td>
<td>output</td>
<td>the enumerated algorithm type.</td>
</tr>
</tbody>
</table>

Status Returned

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**: the library was not initialized.

2.6.14. cusolverRfRefactor()

```c
 cusolverStatus_t cusolverRfRefactor(cusolverRfHandle_t handle);
```

This routine performs the LU re-factorization

\[ A = LU \]

exploring the available parallelism on the GPU. It is assumed that a prior call to the `glu_analyze()` was done in order to find the available parallelism.

This routine may be called multiple times, once for each of the linear systems.
\( A_i x_i = f_i \)

There are some constraints to the combination of algorithms used for refactorization and solving routines, `cusolverRfRefactor()` and `cusolverRfSolve()`. The wrong combination generates the error code `CUSOLVER_STATUS_INVALID_VALUE`. The table below summarizes the supported combinations of algorithms:

**Compatible algorithms for solving and refactorization routines.**

<table>
<thead>
<tr>
<th>Factorization</th>
<th>Solving</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CUSOLVERRF_FACTORIZATION_ALGO0</code></td>
<td><code>TRIANGULAR_SOLVE_ALG1</code></td>
</tr>
<tr>
<td><code>CUSOLVERRF_FACTORIZATION_ALGO1</code></td>
<td><code>TRIANGULAR_SOLVE_ALG2, TRIANGULAR_SOLVE_ALG3</code></td>
</tr>
<tr>
<td><code>CUSOLVERRF_FACTORIZATION_ALGO2</code></td>
<td><code>TRIANGULAR_SOLVE_ALG2, TRIANGULAR_SOLVE_ALG3</code></td>
</tr>
</tbody>
</table>

**Parameter**

- `handle`: Memory, In/out, Meaning
  - Host: the handle to the cuSolverRF library.

**Status Returned**

- `CUSOLVER_STATUS_SUCCESS`: the operation completed successfully.
- `CUSOLVER_STATUS_NOT_INITIALIZED`: the library was not initialized.
- `CUSOLVER_STATUS_EXECUTION_FAILED`: a kernel failed to launch on the GPU.
- `CUSOLVER_STATUS_ZERO_PIVOT`: a zero pivot was encountered during the computation.

### 2.6.15. `cusolverRfResetValues()`

```c
 cusolverStatus_t
 cusolverRfResetValues(/* Input (in the device memory) */
   int n,
   int nnzA,
   int* csrRowPtrA,
   int* csrColIndA,
   double* csrValA,
   int* P,
   int* Q,
   /* Output */
   cusolverRfHandle_t handle);
```

This routine updates internal data structures with the values of the new coefficient matrix. It is assumed that the arrays `csrRowPtrA`, `csrColIndA`, `P` and `Q` have not changed since the last call to the `cusolverRfSetup[Host|Device]` routine. This assumption reflects the fact that the sparsity pattern of coefficient matrices as well as reordering to minimize fill-in and pivoting remain the same in the set of linear systems:

\( A_i x_i = f_i \)

This routine may be called multiple times, once for each of the linear systems.
\[ A_i x_i = f_i \]

<table>
<thead>
<tr>
<th>Parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{nnzA} )</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{csrRowPtrA} )</td>
<td>device</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays ( \text{csrColIndA} ) and ( \text{csrValA} ). This array has also an extra entry at the end that stores the number of non-zero elements in the matrix. The array size is ( n+1 ).</td>
</tr>
<tr>
<td>( \text{csrColIndA} )</td>
<td>device</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is ( \text{nnzA} ).</td>
</tr>
<tr>
<td>( \text{csrValA} )</td>
<td>device</td>
<td>input</td>
<td>the array of values corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is ( \text{nnzA} ).</td>
</tr>
<tr>
<td>( P )</td>
<td>device</td>
<td>input</td>
<td>the left permutation (often associated with pivoting). The array size in ( n ).</td>
</tr>
<tr>
<td>( Q )</td>
<td>device</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in ( n ).</td>
</tr>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**
  
  the operation completed successfully.

- **CUSOLVER_STATUS_NOT_INITIALIZED**
  
  the library was not initialized.

- **CUSOLVER_STATUS_INVALID_VALUE**
  
  an unsupported value or parameter was passed.

- **CUSOLVER_STATUS_EXECUTION_FAILED**
  
  a kernel failed to launch on the GPU.

# 2.6.16. cusolverRfSetMatrixFormat()

cusolverStatus_t

cusolverRfSetMatrixFormat(cusolverRfHandle_t handle,
                          gluMatrixFormat_t format,
                          gluUnitDiagonal_t diag);

This routine sets the matrix format used in the cusolverRfSetupDevice(),
 cusolverRfSetupHost(), cusolverRfResetValues(),
 cusolverRfExtractBundledFactorsHost() and cusolverRfExtractSplitFactorsHost() routines. It may be called once prior to cusolverRfSetupDevice() and
cusolverRfSetupHost() routines.
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>format</td>
<td>host</td>
<td>input</td>
<td>the enumerated matrix format type.</td>
</tr>
<tr>
<td>diag</td>
<td>host</td>
<td>input</td>
<td>the enumerated unit diagonal type.</td>
</tr>
</tbody>
</table>

**Status Returned**

| CUSOLVER_STATUS_SUCCESS              | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED      | the library was not initialized.      |
| CUSOLVER_STATUS_INVALID_VALUE       | an enumerated mode parameter is wrong.|

### 2.6.17. cusolverRfSetNumericProperties()

cusolverStatus_t

cusolverRfSetNumericProperties(cusolverRfHandle_t handle, double zero, double boost);

This routine sets the numeric values used for checking for “zero” pivot and for boosting it in the cusolverRfRefactor() and cusolverRfSolve() routines. It may be called multiple times prior to cusolverRfRefactor() and cusolverRfSolve() routines. The numeric boosting will be used only if boost > 0.0.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>zero</td>
<td>host</td>
<td>input</td>
<td>the value below which zero pivot is flagged.</td>
</tr>
<tr>
<td>boost</td>
<td>host</td>
<td>input</td>
<td>the value which is substituted for zero pivot (if the later is flagged).</td>
</tr>
</tbody>
</table>

**Status Returned**

| CUSOLVER_STATUS_SUCCESS              | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED      | the library was not initialized.      |

### 2.6.18. cusolverRfSetResetValuesFastMode()

cusolverStatus_t

cusolverRfSetResetValuesFastMode(cusolverRfHandle_t handle, gluResetValuesFastMode_t fastMode);

This routine sets the mode used in the cusolverRfResetValues routine. The fast mode requires extra memory and is recommended only if very fast calls to cusolverRfResetValues() are needed. It may be called once prior to cusolverRfAnalyze() routine.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>
fastMode | host | input | the enumerated mode type.

**Status Returned**

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully.
|-------------------------|----------------------------------|
| CUSOLVER_STATUS_NOT_INITIALIZED | the library was not initialized.
| CUSOLVER_STATUS_INVALID_VALUE | an enumerated mode parameter is wrong.

### 2.6.19. cusolverRfSetAlgs()

```c
cusolverStatus_t
cusolverRfSetAlgs(cusolverRfHandle_t handle,
                 gluFactorization_t fact_alg,
                 gluTriangularSolve_t alg);
```

This routine sets the algorithm used for the refactorization in `cusolverRfRefactor()` and the triangular solve in `cusolverRfSolve()`. It may be called once prior to `cusolverRfAnalyze()` routine.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>alg</td>
<td>host</td>
<td>input</td>
<td>the enumerated algorithm type.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>

### 2.6.20. cusolverRfSolve()

```c
cusolverStatus_t
cusolverRfSolve(/* Input (in the device memory) */
                 cusolverRfHandle_t handle,
                 int *P,
                 int *Q,
                 int nrhs,
                 double *Temp,
                 int ldt,
                 /* Input/Output (in the device memory) */
                 double *XF,
                 /* Input */
                 int ldxf);
```

This routine performs the forward and backward solve with the lower $L \in \mathbb{R}^{n \times n}$ and upper $U \in \mathbb{R}^{n \times n}$ triangular factors resulting from the LU re-factorization

$$A = L^* U$$

which is assumed to have been computed by a prior call to the `cusolverRfRefactor()` routine.
The routine can solve linear systems with multiple right-hand-sides (rhs),

\[ AX = (LU)X = L(UX) = LY = F \text{ where } UX = Y \]

even though currently only a single rhs is supported.

This routine may be called multiple times, once for each of the linear systems

\[ A_i x_i = f_i \]

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>input</td>
<td>the left permutation (often associated with pivoting). The array size in ( n ).</td>
</tr>
<tr>
<td>Q</td>
<td>device</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in ( n ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>the number right-hand-sides to be solved.</td>
</tr>
<tr>
<td>Temp</td>
<td>host</td>
<td>input</td>
<td>the dense matrix that contains temporary workspace [of size ( ldt \times \text{nrhs} )].</td>
</tr>
<tr>
<td>ldt</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix ( \text{Temp} ) [( ldt \geq n )].</td>
</tr>
<tr>
<td>XF</td>
<td>host</td>
<td>in/out</td>
<td>the dense matrix that contains the right-hand-sides ( F ) and solutions ( X ) [of size ( ldxf \times \text{nrhs} )].</td>
</tr>
<tr>
<td>ldxf</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix ( \text{XF} ) [( ldxf \geq n )].</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**
  the operation completed successfully.
- **CUSOLVER_STATUS_NOT_INITIALIZED**
  the library was not initialized.
- **CUSOLVER_STATUS_INVALID_VALUE**
  an unsupported value or parameter was passed.
- **CUSOLVER_STATUS_EXECUTION_FAILED**
  a kernel failed to launch on the GPU.
- **CUSOLVER_STATUS_INTERNAL_ERROR**
  an internal operation failed.

### 2.6.21. cusolverRfBatchSetupHost()

```c
 cusolverStatus_t
cusolverRfBatchSetupHost(/* Input (in the host memory) */
   int batchSize,
   int n,
   int nnzA,
   int* h_csrRowPtrA,
   int* h_csrColIndA,
   double* h_csrValA_array[],
   int nnzL,
   int* h_csrRowPtrL,
   int* h_csrColIndL,
   double* h_csrValL,
```

This routine assembles the internal data structures of the cuSolverRF library for batched operation. It is called after the call to the `cusolverRfCreate()` routine, and before any other batched routines.

The batched operation assumes that the user has the following linear systems

\[ A_j x_j = b_j, \quad j = 1, 2, \ldots, \text{batchSize} \]

where each matrix in the set \( \{A_j\} \) has the same sparsity pattern, and quite similar such that factorization can be done by the same permutation \( P \) and \( Q \). In other words, \( A_j, j > 1 \) is a small perturbation of \( A_1 \).

This routine accepts as input (on the host) the original matrix \( A \) (sparsity pattern and batched values), the lower (\( L \)) and upper (\( U \)) triangular factors, as well as the left (\( P \)) and the right (\( Q \)) permutations resulting from the full LU factorization of the first (\( i = 1 \)) linear system

\[ A_1 x_1 = f_1 \]

The permutations \( P \) and \( Q \) represent the final composition of all the left and right reorderings applied to the original matrix \( A \), respectively. However, these permutations are often associated with partial pivoting and reordering to minimize fill-in, respectively.

Remark 1: the matrices \( A, L \) and \( U \) must be CSR format and base-0.

Remark 2: to get best performance, \( \text{batchSize} \) should be multiple of 32 and greater or equal to 32. The algorithm is memory-bound, once bandwidth limit is reached, there is no room to improve performance by large \( \text{batchSize} \). In practice, \( \text{batchSize} \) of 32 - 128 is often enough to obtain good performance, but in some cases larger \( \text{batchSize} \) might be beneficial.

This routine needs to be called only once for a single linear system

\[ A_i x_i = f_i \]
Using the CUSOLVER API

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_csrColIndA</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is n+1.</td>
</tr>
<tr>
<td>h_csrValA_array</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>array of pointers of size batchSize, each pointer points to the array of values corresponding to the non-zero elements in the matrix.</td>
</tr>
<tr>
<td>nnzL</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the number of non-zero elements of matrix L.</td>
</tr>
<tr>
<td>h_csrRowPtrL</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of offsets corresponding to the start of each row in the arrays h_csrColIndL and h_csrValL. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix L. The array size is n+1.</td>
</tr>
<tr>
<td>h_csrColIndL</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of column indices corresponding to the non-zero elements in the matrix L. It is assumed that this array is sorted by row and by column within each row. The array size is n+1.</td>
</tr>
<tr>
<td>h_csrValL</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of values corresponding to the non-zero elements in the matrix L. It is assumed that this array is sorted by row and by column within each row. The array size is nnzL.</td>
</tr>
<tr>
<td>nnzU</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the number of non-zero elements of matrix U.</td>
</tr>
<tr>
<td>h_csrRowPtrU</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of offsets corresponding to the start of each row in the arrays h_csrColIndU and h_csrValU. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix U. The array size is n+1.</td>
</tr>
<tr>
<td>h_csrColIndU</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of column indices corresponding to the non-zero elements in the matrix U. It is assumed that this array is sorted by row and by column within each row. The array size is nnzU.</td>
</tr>
<tr>
<td>h_csrValU</td>
<td>host</td>
<td>input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the array of values corresponding to the non-zero elements in the matrix U. It is assumed that this array is sorted by row and by column within each row. The array size is nnzU.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

### cuSOLVER Library

DU-06709-001_v11.1   |   265

<table>
<thead>
<tr>
<th>h_P</th>
<th>host</th>
<th>input</th>
<th>the left permutation (often associated with pivoting). The array size in n.</th>
</tr>
</thead>
<tbody>
<tr>
<td>h_Q</td>
<td>host</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in n.</td>
</tr>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Status Returned</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_EXECUTION_FAILED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
</tr>
</tbody>
</table>

### 2.6.22. cusolverRfBatchAnalyze()

cusolverStatus_t cusolverRfBatchAnalyze(cusolverRfHandle_t handle);

This routine performs the appropriate analysis of parallelism available in the batched LU refactorization.

It is assumed that a prior call to the cusolverRfBatchSetup[Host]() was done in order to create internal data structures needed for the analysis.

This routine needs to be called only once for a single linear system

\[ A_jx_j = b_j, \ j = 1,2,\ldots, \text{batchSize} \]

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>in/out</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Status Returned</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_EXECUTION_FAILED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
</tr>
</tbody>
</table>

### 2.6.23. cusolverRfBatchResetValues()

cusolverStatus_t

cusolverRfBatchResetValues(/* Input (in the device memory) */
   int batchSize,
   int n,
   int nnzA,
   int* csrRowPtrA,

---

cuSOLVER Library
This routine updates internal data structures with the values of the new coefficient matrix. It is assumed that the arrays csrRowPtrA, csrColIndA, P and Q have not changed since the last call to the cusolverRfbatch_setup_host routine. This assumption reflects the fact that the sparsity pattern of coefficient matrices as well as reordering to minimize fill-in and pivoting remain the same in the set of linear systems

\[ A_j x_j = b_j, \ j = 1,2,..., \text{batchSize} \]

The input parameter csrValA_array is an array of pointers on device memory. csrValA_array(j) points to matrix \( A_j \) which is also on device memory.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>batchSize</td>
<td>host</td>
<td>input</td>
<td>the number of matrices in batched mode.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix A.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays csrColIndA and csrValA. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix. The array size is ( n+1 ).</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>input</td>
<td>the array of column indices corresponding to the non-zero elements in the matrix. It is assumed that this array is sorted by row and by column within each row. The array size is ( nnzA ).</td>
</tr>
<tr>
<td>csrValA_array</td>
<td>device</td>
<td>input</td>
<td>array of pointers of size batchSize, each pointer points to the array of values corresponding to the non-zero elements in the matrix.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>input</td>
<td>the left permutation (often associated with pivoting). The array size in ( n ).</td>
</tr>
<tr>
<td>Q</td>
<td>device</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in ( n ).</td>
</tr>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
</tbody>
</table>
Using the CUSOLVER API

CUSOLVER_STATUS_INVALID_VALUE  |  an unsupported value or parameter was passed.
CUSOLVER_STATUS_EXECUTION_FAILED |  a kernel failed to launch on the GPU.

2.6.24. cusolverRfBatchRefactor()

cusolverStatus_t cusolverRfBatchRefactor(cusolverRfHandle_t handle);

This routine performs the LU re-factorization

\[ M_j = P_j * A_j * Q_j^T = L_j * U_j \]

exploring the available parallelism on the GPU. It is assumed that a prior call to the cusolverRfBatchAnalyze() was done in order to find the available parallelism.

Remark: cusolverRfBatchRefactor() would not report any failure of LU refactorization. The user has to call cusolverRfBatchZeroPivot() to know which matrix failed the LU refactorization.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>in/out</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
</tbody>
</table>

Status Returned

| CUSOLVER_STATUS_SUCCESS         | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED | the library was not initialized.       |
| CUSOLVER_STATUS_EXECUTION_FAILED | a kernel failed to launch on the GPU.    |

2.6.25. cusolverRfBatchSolve()

cusolverStatus_t

cusolverRfBatchSolve( /* Input (in the device memory) */
    cusolverRfHandle_t handle,
    int *P,
    int *Q,
    int nrhs,
    double *Temp,
    int ldt,
    /* Input/Output (in the device memory) */
    double *XF_array[],
    /* Input */
    int ldxf);

To solve \( A_j * x_j = b_j \), first we reform the equation by \( M_j * Q_j * x_j = P_j * b_j \) where \( M_j = P_j * A_j * Q_j^T \). Then do refactorization \( M_j = L_j * U_j \) by cusolverRfBatch_Refactor(). Further cusolverRfBatch_Solve() takes over the remaining steps, including:

\[ z_j = P_j * b_j \]
\[ M_j * y_j = z_j \]
\[ x_j = Q_j^T * y_j \]
The input parameter \( \text{XF\_array} \) is an array of pointers on device memory. \( \text{XF\_array}(j) \) points to matrix \( x_j \) which is also on device memory.

Remark 1: only a single rhs is supported.

Remark 2: no singularity is reported during backward solve. If some matrix \( A_j \) failed the refactorization and \( U_j \) has some zero diagonal, backward solve would compute NaN. The user has to call \text{cusolverRfBatch\_Zero\_Pivot} to check if refactorization is successful or not.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>output</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>input</td>
<td>the left permutation (often associated with pivoting). The array size in ( n ).</td>
</tr>
<tr>
<td>Q</td>
<td>device</td>
<td>input</td>
<td>the right permutation (often associated with reordering). The array size in ( n ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>host</td>
<td>input</td>
<td>the number right-hand-sides to be solved.</td>
</tr>
<tr>
<td>Temp</td>
<td>host</td>
<td>input</td>
<td>the dense matrix that contains temporary workspace (of size ( ldt*nrhs )).</td>
</tr>
<tr>
<td>ldt</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix Temp (( ldt &gt;= n )).</td>
</tr>
<tr>
<td>XF_array</td>
<td>device</td>
<td>in/out</td>
<td>array of pointers of size batchSize, each pointer points to the dense matrix that contains the right-hand-sides F and solutions X (of size ( ldxf*nrhs )).</td>
</tr>
<tr>
<td>ldxf</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix XF (( ldxf &gt;= n )).</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>an unsupported value or parameter was passed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

**2.6.26. cusolverRfBatchZeroPivot()**

```c

cusolverStatus_t
cusolverRfBatchZeroPivot(/* Input */
cusolverRfHandle_t handle
/* Output (in the host memory) */
int *position);
```

Although \( A_j \) is close to each other, it does not mean \( M_j = P^*A_j^*Q^T = L_j^*U_j \) exists for every \( j \). The user can query which matrix failed LU refactorization by checking corresponding value in position array. The input parameter position is an integer array of size batchSize.
The $j$-th component denotes the refactorization result of matrix $A_j$. If $\text{position}(j)$ is -1, the LU refactorization of matrix $A_j$ is successful. If $\text{position}(j)$ is $k \geq 0$, matrix $A_j$ is not LU factorizable and its matrix $U_{j}(i,j)$ is zero.

The return value of `cusolverRfBatch_Zero_Pivot` is CUSOLVER_STATUS_ZERO_PIVOT if there exists one $A_j$ which failed LU refactorization. The user can redo LU factorization to get new permutation $P$ and $Q$ if error code CUSOLVER_STATUS_ZERO_PIVOT is returned.

<table>
<thead>
<tr>
<th>parameter</th>
<th>MemSpace</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>the handle to the cuSolverRF library.</td>
</tr>
<tr>
<td>position</td>
<td>host</td>
<td>output</td>
<td>integer array of size batchSize. The value of position($j$) reports singularity of matrix $A_j$. -1 if no structural/numerical zero, $k \geq 0$ if $A_j(k,k)$ is either structural zero or numerical zero.</td>
</tr>
</tbody>
</table>

Status Returned

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ZERO_PIVOT</td>
<td>a zero pivot was encountered during the computation.</td>
</tr>
</tbody>
</table>
3.1. General description

This chapter describes how to use the cuSolverMG library API. It is not a reference for the cuSolverMG API data types and functions; that is provided in subsequent chapters.

3.1.1. Thread Safety

The library is thread-safe only if one cuSolverMG context per thread.

3.1.2. Determinism

Currently all cuSolverMG API routines from a given toolkit version, generate the same bit-wise results when the following conditions are respected:

- all GPUs particating to the computation have the same compute-capabilities and the same number of SMs.
- the tiles size is kept the same between run.
- number of logical GPUs is kept the same. The order of GPUs are not important because all have the same compute-capabilities.

3.1.3. tile strategy

The tiling strategy of cuSolverMG is compatible with ScaLAPACK. The current release only supports 1-D column block cyclic, column-major PACKED format.

Figure 1.a shows a partition of the matrix $A$ of dimension $M_A \times N_A$. Each column tile has $T_A$ columns. There are seven columns tiles, labeled as $0,1,2,3,4,5,6$, distributed into three GPUs in a cyclic way, i.e. each GPU takes one column tile in turn. For example, GPU 0 has column tile $0, 3, 6$ (yellow tiles) and GPU 1 takes column tiles next to GPU 0 (blue tiles). Not all GPUs have the same number of tiles, in this example, GPU 0 has three tiles, others have only two tiles.

Figure 1.b shows two possible formats to store those column tiles locally in each GPU. Left side is called PACKED format and right side is UNPACKED format. PACKED format aggregates
three column tiles in a contiguous memory block while UNPACKED format distributes these three column tiles into different memory blocks. The only difference between them is that PACKED format can have a big GEMM call instead of three GEMM calls in UNPACKED format. So theoretically speaking, PACKED format can deliver better performance than UNPACKED format. cuSolverMG only supports PACKED format in the API. In order to achieve maximal performance, the user just needs to choose proper tile size \( T_A \) to partition the matrix, not too small, for example 256 or above is enough.

There is another parameter, called \( LLD_A \), to control the leading dimension of the local matrix in each GPU. \( LLD_A \) must be greater or equal to \( M_A \). The purpose of \( LLD_A \) is for better performance of GEMM. For small problem, GEMM is faster if \( LLD_A \) is power of 2. However for big problem, \( LLD_A \) does not show significant improvement. cuSolverMG only supports \( LLD_A=M_A \).

Figure 1. Example of cuSolverMG tiling for 3 Gpus

The processing grid in cuSolverMG is a list of GPU IDs, similar to the process ID in ScaLAPACK. cuSolverMG only supports 1D column block cyclic, so only 1D grid is supported as well. Suppose \( deviceId \) is a list of GPU IDs, both \( deviceId=1,1,1 \) and \( deviceId=2,1,0 \) are valid. The former describes three logical devices are selected to run cuSolverMG routines, and all have the same physical ID, 0. The latter still uses three logical devices, but each has different physical ID. The current design only accepts 32 logical devices, that is, the length of \( deviceId \) is less or equal to 32. Figure 1 uses \( deviceId=0,1,2 \).

In practice, the matrix \( A \) is distributed into GPUs listed in \( deviceId \). If the user chooses \( deviceId=1,1,1 \), all columns tile are located in GPU 1, this will limit the size of the problem because of memory capacity of one GPU. Besides, multiGPU routine adds extra overhead on data communication through off-chip bus, which has big performance impact if NVLINK is not supported or used. It would be faster to run on single GPU instead of running multiGPU version with devices of the same GPU ID.

3.1.4. Global matrix versus local matrix

To operate a submatrix of the matrix \( A \) is simple in dense linear algebra, just shift the pointer to the starting point of the submatrix relative to \( A \). For example, \( \text{gesvd}(10,10, A) \) is SVD
of $A(0:9,0:9). \text{gesvd}(10,10, A + 5 + 2*da)$ is SVD of 10-by-10 submatrix starting at $A(5,2)$.

However it is not simple to operate on a submatrix of a distributed matrix because different starting point of the submatrix changes the distribution of the layout of that submatrix. ScaLAPACK introduces two parameters, $IA$ and $JA$, to locate the submatrix. Figure 2 shows (global) matrix $A$ of dimension $M_A$ by $N_A$. The $\text{sub}(A)$ is a $M$ by $N$ submatrix of $A$, starting at $IA$ and $JA$. Please be aware that $IA$ and $JA$ are base-1.

Given a distributed matrix $A$, the user can compute eigenvalues of the submatrix $\text{sub}(A)$ by either calling $\text{syevd}(A, IA, JA)$ or gathering $\text{sub}(A)$ to another distributed matrix $B$ and calling $\text{syevd}(B, IB=1, JB=1)$.

**Figure 2.** global matrix and local matrix

3.1.5. usage of _BufferSize

There is no cudaMalloc inside cuSolverMG library, the user must allocate the device workspace explicitly. The routine $\text{xyz_bufferSize}$ is to query the size of workspace of the routine $\text{xyz}$, for example $\text{xyz} = \text{syevd}$. To make the API simple, $\text{xyz_bufferSize}$ follows almost the same signature of $\text{xyz}$ even it only depends on some parameters, for example, device pointer is not used to decide the size of workspace. In most cases, $\text{xyz_bufferSize}$ is called in the beginning before actual device data (pointing by a device pointer) is prepared or before the device pointer is allocated. In such case, the user can pass null pointer to $\text{xyz_bufferSize}$ without breaking the functionality.

$\text{xyz_bufferSize}$ returns $\text{bufferSize}$ for each device. The size is number of elements, not number of bytes.
3.1.6. synchronization

All routines are in synchronous (blocking call) manner. The data is ready after the routine. However the user has to prepare the distributed data before calling the routine. For example, if the user has multiple streams to setup the matrix, stream synchronization or device synchronization is necessary to guarantee distributed matrix is ready.

3.1.7. context switch

The user does not need to restore the device by `cudaSetDevice()` after each cuSolverMG call. All routines set the device back to what the caller has.

3.1.8. NVLINK

The peer-to-peer communication via NVLINK can dramatically reduce the overhead of data exchange among GPUs. `cuSolverMG` does not enable NVLINK implicitly, instead, it gives this option back to the user, not to interfere other libraries. The example code H.1 shows how to enable peer-to-peer communication.

3.2. cuSolverMG Types Reference

3.2.1. cuSolverMG Types

The `float`, `double`, `cuComplex`, and `cuDoubleComplex` data types are supported. The first two are standard C data types, while the last two are exported from `cuComplex.h`. In addition, `cuSolverMG` uses some familiar types from `cuBlas`.

3.2.2. cusolverMgHandle_t

This is a pointer type to an opaque `cuSolverMG` context, in which the user must initialize by calling `cusolverMgCreate()` prior to calling any other library function. An un-initialized handle object will lead to unexpected behavior, including crashes of `cuSolverMG`. The handle created and returned by `cusolverMgCreate()` must be passed to every `cuSolverMG` function.

3.2.3. cusolverMgGridMapping_t

The type indicates layout of grids.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDALIBMG_GRID_MAPPING_ROW_MAJOR</td>
<td>row-major ordering.</td>
</tr>
<tr>
<td>CUDALIBMG_GRID_MAPPING_COL_MAJOR</td>
<td>column-major ordering.</td>
</tr>
</tbody>
</table>

3.2.4. cudaLibMgGrid_t

Opaque structure of the distributed grid.
3.2.5. **cudaLibMgMatrixDesc_t**

opaque structure of the distributed matrix descriptor.

### 3.3. Helper Function Reference

#### 3.3.1. **cusolverMgCreate()**

```c
cusolverStatus_t
cusolverMgCreate(cusolverMgHandle_t *handle)
```

This function initializes the cuSolverMG library and creates a handle on the cuSolverMG context. It must be called before any other cuSolverMG API function is invoked. It allocates hardware resources necessary for accessing the GPU.

**Output**

<table>
<thead>
<tr>
<th>handle</th>
<th>the pointer to the handle to the cuSolverMG context.</th>
</tr>
</thead>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the initialization succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
</tbody>
</table>

#### 3.3.2. **cusolverMgDestroy()**

```c
cusolverStatus_t
cusolverMgDestroy( cusolverMgHandle_t handle)
```

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

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>the handle to the cuSolverMG context.</th>
</tr>
</thead>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the shutdown succeeded.</th>
</tr>
</thead>
</table>

#### 3.3.3. **cusolverMgDeviceSelect()**

```c
cusolverStatus_t
cusolverMgDeviceSelect ( 
    cusolverMgHandle_t handle,  
    int nbDevices,  
    int deviceId[] )
```

This function registers a subset of devices (GPUs) to cuSolverMG handle. Such subset of devices is used in subsequent API calls. The array deviceId contains a list of logical device ID.
The term logical means repeated device ID are permitted. For example, suppose the user has only one GPU in the system, say device 0, if he sets deviceId=0,0,0, then cuSolverMG treats them as three independent GPUs, one stream each, so concurrent kernel launches still hold. The current design only supports up to 32 logical devices.

**Input**

<table>
<thead>
<tr>
<th>Input</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>the pointer to the handle to the cuSolverMG context.</td>
</tr>
<tr>
<td>nbDevices</td>
<td>the number of logical devices</td>
</tr>
<tr>
<td>deviceId</td>
<td>an integer array of size nbDevices</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the initialization succeeded.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>nbDevices must be greater than zero, and less or equal to 32.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>internal error occurs when setting internal streams and events.</td>
</tr>
</tbody>
</table>

### 3.3.4. cusolverMgCreateDeviceGrid()

cusolverStatus_t cusolverMgCreateDeviceGrid(  
cusolverMgGrid_t* grid,  
int32_t numRowDevices,  
int32_t numColDevices,  
const int32_t deviceId[],  
cusolverMgGridMapping_t mapping)

This function setups grid of devices.

Only 1-D column block cyclic is supported, so numRowDevices must be equal to 1.

**WARNING:** cusolverMgCreateDeviceGrid() must be consistent with cusolverMgDeviceSelect(), i.e. numColDevices must be equal to nbDevices in cusolverMgDeviceSelect().

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>host</td>
<td>output</td>
<td>the pointer to the opaque structure.</td>
</tr>
<tr>
<td>numRowDevices</td>
<td>host</td>
<td>input</td>
<td>number of devices in the row.</td>
</tr>
<tr>
<td>numColDevices</td>
<td>host</td>
<td>input</td>
<td>number of devices in the column.</td>
</tr>
<tr>
<td>deviceId</td>
<td>host</td>
<td>input</td>
<td>integer array of size numColDevices, containing device IDs.</td>
</tr>
<tr>
<td>mapping</td>
<td>host</td>
<td>input</td>
<td>row-major or column-major ordering.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Returned</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>numColDevices is not greater than 0.</td>
</tr>
<tr>
<td></td>
<td>numRowDevices is not 1.</td>
</tr>
</tbody>
</table>
3.3.5. cusolverMgDestroyGrid()

```c
 cusolverStatus_t
cusolverMgDestroyGrid(
    cusolverMgGrid_t grid)
```

This function releases resources of a grid.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid</td>
<td>host</td>
<td>input/output</td>
<td>the pointer to the opaque structure.</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.

3.3.6. cusolverMgCreateMatDescr()

```c
 cusolverStatus_t
cusolverMgCreateMatrixDesc(
    cusolverMgMatrixDesc_t * desc,
    int64_t numRows, int64_t numCols, int64_t rowBlockSize, int64_t colBlockSize,
    cudaDataType_t dataType, const cusolverMgGrid_t grid)
```

This function setups the matrix descriptor desc.

Only 1-D column block cyclic is supported, so numRows must be equal to rowBlockSize.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>host</td>
<td>output</td>
<td>the matrix descriptor.</td>
</tr>
<tr>
<td>numRows</td>
<td>host</td>
<td>input</td>
<td>the number of rows of global A.</td>
</tr>
<tr>
<td>numCols</td>
<td>host</td>
<td>input</td>
<td>the number of columns of global A.</td>
</tr>
<tr>
<td>rowBlockSize</td>
<td>host</td>
<td>input</td>
<td>the number of rows per tile.</td>
</tr>
<tr>
<td>colBlockSize</td>
<td>host</td>
<td>input</td>
<td>the number of columns per tile.</td>
</tr>
<tr>
<td>dataType</td>
<td>host</td>
<td>input</td>
<td>data type of the matrix.</td>
</tr>
<tr>
<td>grid</td>
<td>host</td>
<td>input</td>
<td>the pointer to structure of grid.</td>
</tr>
</tbody>
</table>

**Status Returned**

- **CUSOLVER_STATUS_SUCCESS**: the operation completed successfully.
- **CUSOLVER_STATUS_INVALID_VALUE**: numRows, numCols, or rowBlockSize or colBlockSize is less than 0. numRows is not equal to rowBlockSize.

3.3.7. cusolverMgDestroyMatrixDescr()
cusolverMgDestroyMatrixDesc(
    cusolverMgMatrixDesc_t desc)
This function releases the matrix descriptor desc.

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>host</td>
<td>input/output</td>
<td>the matrix descriptor.</td>
</tr>
</tbody>
</table>

**Status Returned**

| CUSOLVER_STATUS_SUCCESS | the operation completed successfully. |

### 3.4. Dense Linear Solver Reference

This chapter describes linear solver API of cuSolverMG.

#### 3.4.1. cusolverMgPotrf()

The helper function below can calculate the sizes needed for pre-allocated buffer for cusolverMgPotrf.

```c
cusolverStatus_t
cusolverMgPotrf_bufferSize(
    cusolverMgHandle_t handle,
    cublasFillMode_t uplo,
    int N,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    cudaDataType computeType,
    int64_t *lwork)
```

The routine bellow

```c
cusolverStatus_t
cusolverMgPotrf(
    cusolverMgHandle_t handle,
    cublasFillMode_t uplo,
    int N,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    cudaDataType computeType,
    void *array_d_work[],
    int64_t lwork,
    int *info)
```

computes the Cholesky factorization of a Hermitian positive-definite matrix using the generic API interfacete.

A is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful. The input parameter \( \text{uplo} \) indicates which part of the matrix is used. The function would leave other part untouched.
If input parameter `uplo` is `CUBLAS_FILL_MODE_LOWER`, only lower triangular part of \( A \) is processed, and replaced by lower triangular Cholesky factor \( L \).

\[
A = L^* L^H
\]

If input parameter `uplo` is `CUBLAS_FILL_MODE_UPPER`, only upper triangular part of \( A \) is processed, and replaced by upper triangular Cholesky factor \( U \).

\[
A = U^{H*} U
\]

The user has to provide device working space in `array_d_work`. `array_d_work` is a host pointer array of dimension \( G \), where \( G \) is number of devices. `array_d_work[j]` is a device pointer pointing to a device memory in \( j \)-th device. The data type of `array_d_work[j]` is `computeType`. The size of `array_d_work[j]` is `lwork` which is number of elements per device, returned by `cusolverMgPotrf_bufferSize()`.

If Cholesky factorization failed, i.e. some leading minor of \( A \) is not positive definite, or equivalently some diagonal elements of \( L \) or \( U \) is not a real number. The output parameter `info` would indicate smallest leading minor of \( A \) which is not positive definite.

If output parameter `info` = -\( i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

The generic API has two different types, `dataTypeA` is data type of the matrix \( A \), `computeType` is compute type of the operation and data type of the workspace `[array_d_work]` contains `dataTypeA`, so there is no explicit parameter of `dataTypeA`. `cusolverMgPotrf` only supports the following four combinations.

Appendix I provides an example of `cusolverMgPotrf`.

**Valid combination of data type and compute type**

<table>
<thead>
<tr>
<th><code>dataTypeA</code></th>
<th><code>computeType</code></th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRF</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRF</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRF</td>
</tr>
</tbody>
</table>

**API of potrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverMg library context.</td>
</tr>
<tr>
<td><code>uplo</code></td>
<td>host</td>
<td>input</td>
<td>indicates if matrix ( A ) lower or upper part is stored, the other part is not referenced. Only <code>CUBLAS_FILL_MODE_LOWER</code> is supported.</td>
</tr>
<tr>
<td><code>N</code></td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix <code>sub(A)</code>.</td>
</tr>
<tr>
<td><code>array_d_A</code></td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension ( G ). It contains a distributed &lt;type&gt; array containing <code>sub(A)</code> of dimension ( N \times N ).</td>
</tr>
</tbody>
</table>
On exit, sub(A) contains the factors L or U.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array A indicating the first row of sub(A).</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array A indicating the first column of sub(A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>matrix descriptor for the distributed matrix A.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>array_d_work</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. array_d_work[j] points to a device working space in j-th device, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of array_d_work[j], returned by cusolverMgPotrf_bufferSize. lwork denotes number of elements, not number of bytes.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>if info = 0, the Cholesky factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i, the leading minor of order i is not positive definite.</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_SUCCESS</th>
<th>the operation completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [M, N&lt;0].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

### 3.4.2. cusolverMgPotrs()

The helper function below can calculate the sizes needed for pre-allocated buffer for cusolverMgPotrs.

```c
cusolverStatus_t
cusolverMgPotrs_bufferSize(
    cusolverMgHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    int nrhs,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    void *array_d_B[],
    int IB,
    int JB,
    cudaLibMgMatrixDesc_t descrB,
```
Using the CUSOLVERMG API

The routine below

```c
 cusolverStatus_t
cusolverMgPotrs(
    cusolverMgHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    int nrhs,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    void *array_d_B[],
    int IB,
    int JB,
    cudaLibMgMatrixDesc_t descrB,
    cudaDataType computeType,
    void *array_d_work[],
    int64_t lwork,
    int *info)
```

This function solves a system of linear equations

\[ A \ast X = B \]

where \( A \) is a \( n \times n \) Hermitian matrix, only lower or upper part is meaningful using the generic API interface. The input parameter \( \text{uplo} \) indicates which part of the matrix is used. The function would leave other part untouched.

If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_LOWER}, the matrix should \( A \) contain the lower triangular factor for Cholesky decomposition previously computed by \text{cusolverMgPotrf} routine.

\[ A = L \ast L^H \]

If input parameter \( \text{uplo} \) is \text{CUBLAS_FILL_MODE_UPPER}, the matrix should \( A \) contain the upper triangular factor for Cholesky decomposition previously computed by \text{cusolverMgPotrf} routine.

\[ A = U^H \ast U \]

The operation is in-place, i.e. matrix \( B \) contains the solution of the linear system on exit.

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).

The user has to provide device working space in \text{array_d_work}. \text{array_d_work} is a host pointer array of dimension \( G \), where \( G \) is number of devices. \text{array_d_work}[j] is a device pointer pointing to a device memory in \( j \)-th device. The data type of \text{array_d_work}[j] is \text{computeType}. The size of \text{array_d_work}[j] is \( lwork \) which is number of elements per device, returned by \text{cusolverMgPotrs_bufferSize}().

If output parameter \( \text{info} = -i \) (less than zero), the \( i \)-th parameter is wrong (not counting handle).
The generic API has four different types, dataTypeA is data type of the matrix A, dataTypeB is data type of the matrix B, computeType is compute type of the operation and data type of the workspace [array_d_work] descrA contains dataTypeA and descrB contains dataTypeB and so there is no explicit parameter of dataTypeA and dataTypeB. cusolverMgPotrs only supports the following four combinations.

Appendix I provides an example of cusolverMgPotrs.

valid combination of data type and compute type

<table>
<thead>
<tr>
<th>DataTypeA</th>
<th>DataTypeB</th>
<th>ComputeType</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRS</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRS</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRS</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRS</td>
</tr>
</tbody>
</table>

API of potrs

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverMg library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix A lower or upper part is stored, the other part is not referenced. Only CUBLAS_FILL_MODE_LOWER is supported.</td>
</tr>
<tr>
<td>N</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix sub(A).</td>
</tr>
<tr>
<td>NRHS</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix sub(A) and sub(B).</td>
</tr>
<tr>
<td>array_d_A</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. It contains a distributed &lt;type&gt; array containing sub(A) of dimension M * N. On exit, sub(A) contains the factors L and U.</td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array A indicating the first row of sub(A).</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array A indicating the first column of sub(A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>matrix descriptor for the distributed matrix A.</td>
</tr>
<tr>
<td>array_d_B</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. It contains a distributed &lt;type&gt; array containing sub(B) of dimension N * NRHS. On exit, sub(A) contains the solution to the linear system.</td>
</tr>
<tr>
<td>IB</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array B indicating the first row of sub(B).</td>
</tr>
<tr>
<td>JB</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array B indicating the first column of sub(B).</td>
</tr>
</tbody>
</table>
Using the CUSOLVERMG API

<table>
<thead>
<tr>
<th>descrB</th>
<th>host</th>
<th>input</th>
<th>matrix descriptor for the distributed matrix B.</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>array_d_work</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>array_d_work[j] points to a device working space in j-th device, &lt;type&gt;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of array_d_work[j], returned by cusolverMgPotrs_bufferSize.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lwork denotes number of elements, not number of bytes.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>if info = 0, the routine successful. if info = -i, the i-th parameter is wrong (not counting handle).</td>
</tr>
</tbody>
</table>

Status Returned

| CUSOLVER_STATUS_SUCCESS          | the operation completed successfully. |
| CUSOLVER_STATUS_NOT_INITIALIZED  | the library was not initialized.      |
| CUSOLVER_STATUS_INVALID_VALUE    | invalid parameters were passed \( M, N < 0 \). |
| CUSOLVER_STATUS_ARCH_MISMATCH    | the device only supports compute capability 2.0 and above. |
| CUSOLVER_STATUS_INTERNAL_ERROR   | an internal operation failed.         |

3.4.3. cusolverMgPotri()

The helper function below can calculate the sizes needed for pre-allocated buffer for cusolverMgPotri.

```c
cusolverStatus_t cusolverMgPotri_bufferSize(
    cusolverMgHandle_t handle,
    cublasFillMode_t `uplo`,
    int N,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    cudaDataType computeType,
    int64_t *lwork)
```

The routine bellow

```c
cusolverStatus_t cusolverMgPotri(
    cusolverMgHandle_t handle,
    cublasFillMode_t `uplo`,
    int N,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    cudaDataType computeType,
```
Using the CUSOLVERMG API

```c
void *array_d_work[],
int64_t lwork,
int *info)
```

This function computes the inverse of an Hermitian positive-definite matrix \( A \) using the Cholesky factorization

\[
A = L^H \times L = U^H \times U
\]

computed by \texttt{cusolverMgPotrf}().

If the input parameter \texttt{uplo} is \texttt{CUBLAS_FILL_MODE_LOWER}, on input, matrix \( A \) contains the lower triangular factor of \( A \) computed by \texttt{cusolverMgPotrf}. Only lower triangular part of \( A \) is processed, and replaced the by lower triangular part of the inverse of \( A \).

If the input parameter \texttt{uplo} is \texttt{CUBLAS_FILL_MODE_UPPER}, on input, matrix \( A \) contains the upper triangular factor of \( A \) computed by \texttt{cusolverMgPotrf}. Only upper triangular part of \( A \) is processed, and replaced the by upper triangular part of the inverse of \( A \).

The user has to provide device working space in \texttt{array_d_work}. \texttt{array_d_work} is a host pointer array of dimension \( G \), where \( G \) is number of devices. \texttt{array_d_work[j]} is a device pointer pointing to a device memory in \( j \)-th device. The data type of \texttt{array_d_work[j]} is \texttt{computeType}. The size of \texttt{array_d_work[j]} is \( lwork \) which is number of elements per device, returned by \texttt{cusolverMgPotri_bufferSize}().

If the computation of the inverse fails, i.e. some leading minor of \( L \) or \( U \), is null, the output parameter \texttt{info} would indicate the smallest leading minor of \( L \) or \( U \) which is not positive definite.

If the output parameter \texttt{info} = -\( i \) (less than zero), the \( i \)-th parameter is wrong (not counting the handle).

The generic API has two different types, \texttt{dataTypeA} is data type of the matrix \( A \), \texttt{computeType} is compute type of the operation and data type of the workspace \texttt{array_d_work}. \texttt{descrA} contains \texttt{dataTypeA}, so there is no explicit parameter of \texttt{dataTypeA}. \texttt{cusolverMgPotri} only supports the following four combinations.

Appendix I provides an example of \texttt{cusolverMgPotri}.

### Valid combination of data type and compute type

<table>
<thead>
<tr>
<th>\texttt{DataTypeA}</th>
<th>\texttt{ComputeType}</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SPOTRI</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DPOTRI</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_C_32F</td>
<td>CPOTRI</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_C_64F</td>
<td>ZPOTRI</td>
</tr>
</tbody>
</table>

### API of \texttt{potrf}

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverMg library context.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>indicates if matrix ( A ) lower or upper part is stored, the other part is not referenced. Only \texttt{CUBLAS_FILL_MODE_LOWER} is supported.</td>
</tr>
</tbody>
</table>

---

\textit{cuSOLVER Library}
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix sub(A).</td>
</tr>
<tr>
<td>array_d_A</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. It contains a distributed &lt;type&gt; array containing sub(A) of dimension N * N. On exit, sub(A) contains the upper or lower triangular part of the inverse of A depending on the value of uplo argument.</td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array A indicating the first row of sub(A).</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array A indicating the first column of sub(A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>matrix descriptor for the distributed matrix A.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>array_d_work</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. array_d_work[j] points to a device working space in j-th device, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of array_d_work[j], returned by cusolverMgPotri_bufferSize. lwork denotes number of elements, not number of bytes.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>if info = 0, the Cholesky factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i, the leading minor of order i is zero.</td>
</tr>
</tbody>
</table>

### Status Returned

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed [M, N&lt;0].</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

#### 3.4.4. cusolverMgGetrf()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
 cusolverMgGetrf_bufferSize(
   cusolverMgHandle_t handle,
   int M,
   int N,
   void *array_d_A[],
   int IA,
);```

---

Using the CUSOLVERMG API
This function computes the LU factorization of a $M \times N$ matrix

$$P \times A = L \times U$$

where $A$ is a $M \times N$ matrix, $P$ is a permutation matrix, $L$ is a lower triangular matrix with unit diagonal, and $U$ is an upper triangular matrix.

The user has to provide device working space in $array_d_work$. $array_d_work$ is a host pointer array of dimension $G$, where $G$ is number of devices. $array_d_work[j]$ is a device pointer pointing to a device memory in $j$-th device. The data type of $array_d_work[j]$ is $computeType$. The size of $array_d_work[j]$ is $lwork$ which is number of elements per device, returned by $cusolverMgGetrf_bufferSize()$.

If LU factorization failed, i.e. matrix $A$ [U] is singular, The output parameter $info=i$ indicates $U(i,i) = 0$.

If output parameter $info = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle).

If $array_d_IPIV$ is null, no pivoting is performed. The factorization is $A=L\times U$, which is not numerically stable.

$array_d_IPIV$ must be consistent with $array_d_A$, i.e. $JA$ is the first column of $sub(A)$, also the first column of $sub(IPIV)$.

No matter LU factorization failed or not, the output parameter $array_d_IPIV$ contains pivoting sequence, row $i$ is interchanged with row $array_d_IPIV(i)$.

The generic API has three different types, $dataTypeA$ is data type of the matrix $A$, $computeType$ is compute type of the operation and data type of the workspace $[array\_d\_work]$ descrA contains $dataTypeA$, so there is no explicit parameter of $dataTypeA$.

$cusolverMgGetrf$ only supports the following four combinations.

<table>
<thead>
<tr>
<th>valid combination of data type and compute type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dataTypeA$</td>
</tr>
</tbody>
</table>

Appendix I provides an example of $cusolverMgGetrf$. 

Using the CUSOLVERMG API

cusolverStatus_t cusolverMgGetrf(
    cusolverMgHandle_t handle,
    int M,
    int N,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    int *array_d_IPIV[],
    cudaDataType_t computeType,
    void *array_d_work[],
    int64_t lwork,
    int *info );
Using the CUSOLVERMG API

| CUDA_R_32F | CUDA_R_32F | SGETF R |
|CUDA_R_64F | CUDA_R_64F | DGETF R |
|CUDA_C_32F | CUDA_C_32F | CGETF R |
|CUDA_C_64F | CUDA_C_64F | ZGETF R |

Remark 1: tile size TA must be less or equal to 512.

**API of getrf**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverMg library context.</td>
</tr>
<tr>
<td>M</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix sub(A).</td>
</tr>
<tr>
<td>N</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix sub(A).</td>
</tr>
<tr>
<td>array_d_A</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. It contains a distributed &lt;type&gt; array containing sub(A) of dimension M * N. On exit, sub(A) contains the factors L and U.</td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array A indicating the first row of sub(A).</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array A indicating the first column of sub(A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>matrix descriptor for the distributed matrix A.</td>
</tr>
<tr>
<td>array_d_IPIV</td>
<td>host</td>
<td>output</td>
<td>a host pointer array of dimension G. It contains a distributed integer array containing sub(IPIV) of size min(M,N). sub(IPIV) contains pivot indices.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>array_d_work</td>
<td>host</td>
<td>in/out</td>
<td>a host pointer array of dimension G. array_d_work[j] points to a device working space in j-th device, &lt;type&gt; array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>size of array_d_work[j], returned by cusolverMgGetrf_bufferSize. lwork denotes number of elements, not number of bytes.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>if info = 0, the LU factorization is successful. if info = -i, the i-th parameter is wrong (not counting handle). if info = i, the U(i,i) = 0.</td>
</tr>
</tbody>
</table>

**Status Returned**

- CUSOLVER_STATUS_SUCCESS: the operation completed successfully.
- CUSOLVER_STATUS_INVALID_VALUE: invalid parameters were passed [M, N<0].
- CUSOLVER_STATUS_INTERNAL_ERROR: an internal operation failed.
3.4.5. **cusolverMgGetrs()**

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverMgGetrs_bufferSize(
    cusolverMgHandle_t handle,
    cublasOperation_t TRANS,
    int N,
    int NRHS,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    int *array_d_IPIV[],
    void *array_d_B[],
    int IB,
    int JB,
    cudaLibMgMatrixDesc_t descrB,
    cudaDataType_t computeType,
    int64_t *lwork);
```

```c
cusolverStatus_t
cusolverMgGetrs(
    cusolverMgHandle_t handle,
    cublasOperation_t TRANS,
    int N,
    int NRHS,
    void *array_d_A[],
    int IA,
    int JA,
    cudaLibMgMatrixDesc_t descrA,
    int *array_d_IPIV[],
    void *array_d_B[],
    int IB,
    int JB,
    cudaLibMgMatrixDesc_t descrB,
    cudaDataType_t computeType,
    void *array_d_work[],
    int64_t lwork,
    int *info );
```

This function solves a linear system of multiple right-hand sides

\[ \text{op}(A)X = B \]

where \( A \) is a \( N \times N \) matrix, and was LU-factored by \text{getrf} \text{f}, that is, lower triangular part of \( A \) is \( L \), and upper triangular part (including diagonal elements) of \( A \) is \( U \). \( B \) is a \( N \times NRHS \) right-hand side matrix. The solution matrix \( X \) overwrites the right-hand-side matrix \( B \).

The input parameter \( \text{TRANS} \) is defined by

\[
\text{op}(A) = \begin{cases} 
A & \text{if } \text{TRANS} == \text{CUBLAS_OP_N} \\
A^T & \text{if } \text{TRANS} == \text{CUBLAS_OP_T} \\
A^H & \text{if } \text{TRANS} == \text{CUBLAS_OP_C}
\end{cases}
\]
The user has to provide device working space in `array_d_work`. `array_d_work` is a host pointer array of dimension `G`, where `G` is number of devices. `array_d_work[j]` is a device pointer pointing to a device memory in `j`-th device. The data type of `array_d_work[j]` is `computeType`. The size of `array_d_work[j]` is `lwork` which is number of elements per device, returned by `cusolverMgGetrs_bufferSize()`.

If `array_d_IPIV` is null, no pivoting is performed. Otherwise, `array_d_IPIV` is an output of `getrf`. It contains pivot indices, which are used to permute right-hand sides.

If output parameter `info = -i` (less than zero), the `i`-th parameter is wrong (not counting handle).

The generic API has three different types, `dataTypeA` is data type of the matrix `A`, `dataTypeB` is data type of the matrix `B`, and `computeType` is compute type of the operation and data type of the workspace `{array_d_work}`. `descrA` contains `dataTypeA`, so there is no explicit parameter of `dataTypeA`. `descrB` contains `dataTypeB`, so there is no explicit parameter of `dataTypeB`. `cusolverMgGetrs` only supports the following four combinations.

<table>
<thead>
<tr>
<th>valid combination of data type and compute type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataTypeA</strong></td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

Remark 1: tile size `TA` must be less or equal to 512.

Remark 2: only support `TRANS=CUBLAS_OP_N`.

Appendix I provides an example of `cusolverMgGetrs`.

**API of `getrs`**

<table>
<thead>
<tr>
<th>parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverMG library context.</td>
</tr>
<tr>
<td>TRANS</td>
<td>host</td>
<td>input</td>
<td>operation <code>op(A)</code> that is non- or (conj.) transpose.</td>
</tr>
<tr>
<td>N</td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix <code>sub(A)</code>.</td>
</tr>
<tr>
<td>NRHS</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix <code>sub(B)</code>.</td>
</tr>
<tr>
<td><code>array_d_A</code></td>
<td>host</td>
<td>input</td>
<td>a host pointer array of dimension <code>G</code>. It contains a distributed <code>&lt;type&gt;</code> array containing <code>sub(A)</code> of dimension <code>M * N</code>. <code>sub(A)</code> contains the factors <code>L</code> and <code>U</code>.</td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array <code>A</code> indicating the first row of <code>sub(A)</code>.</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array <code>A</code> indicating the first column of <code>sub(A)</code>.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>matrix descriptor for the distributed matrix <code>A</code>.</td>
</tr>
</tbody>
</table>
### 3.5. Dense Eigenvalue Solver Reference

This chapter describes eigenvalue solver API of cuSolverMG.

#### 3.5.1. cusolverMgSyevd()

The helper functions below can calculate the sizes needed for pre-allocated buffer.

```c
cusolverStatus_t
cusolverMgSyevd_bufferSize(
    cusolverMgHandle_t handle,
    cusolverEigMode_t jobz,
    cublasFillMode_t uplo,
    int N,
    void *array_d_A[]),
```
using the CUSOLVERMG API

int IA,
int JA,
cudaLibMgMatrixDesc_t descrA,
void *W,
cudaDataType_t dataTypeW,
cudaDataType_t computeType,
int64_t *lwork
);

cusolverStatus_t
cusolverMgSyevd(
    cusolverMgHandle_t handle,
cusolverEigMode_t jobz,
cublasFillMode_t uplo,
    int N,
    void *__array_d_A[ ],
    int IA,
    int JA,
cudaLibMgMatrixDesc_t descrA,
    void *__W,
cudaDataType_t dataTypeW,
cudaDataType_t computeType,
    void *__array_d_work[ ],
    int64_t lwork,
    int *__info );

This function computes eigenvalues and eigenvectors of a symmetric (Hermitian) \( N \times N \) matrix \( A \). The standard symmetric eigenvalue problem is

\[ A \mathbf{V} = \mathbf{V} \Lambda \]

where \( \Lambda \) is a real \( N \times N \) diagonal matrix, \( \mathbf{V} \) is an \( N \times N \) unitary matrix. The diagonal elements of \( \Lambda \) are the eigenvalues of \( A \) in ascending order.

cusolverMgSyevd returns the eigenvalues in \( W \) and overwrites the eigenvectors in \( A \). \( W \) is a host \( 1 \times N \) vector.

The generic API has three different types, \( \text{dataTypeA} \) is data type of the matrix \( A \), \( \text{dataTypeW} \) is data type of the vector \( W \), and \( \text{computeType} \) is compute type of the operation and data type of the workspace \( \text{array_d_work} \). \( \text{descrA} \) contains \( \text{dataTypeA} \), so there is no explicit parameter of \( \text{dataTypeA} \). \( \text{cusolverMgSyevd} \) only supports the following four combinations.

**valid combination of data type and compute type**

<table>
<thead>
<tr>
<th>( \text{DataTypeA} )</th>
<th>( \text{DataTypeW} )</th>
<th>( \text{ComputeType} )</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>SSYEVD</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>DSYEVD</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_C_32F</td>
<td>CHEEVD</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_C_64F</td>
<td>ZHEEVD</td>
</tr>
</tbody>
</table>

The user has to provide device working space in \( \text{array_d_work} \). \( \text{array_d_work} \) is a host pointer array of dimension \( G \), where \( G \) is number of devices. \( \text{array_d_work}[j] \) is a device pointer pointing to a device memory in \( j \)-th device. The data type of \( \text{array_d_work}[j] \) is \( \text{computeType} \). The size of \( \text{array_d_work}[j] \) is \( \text{lwork} \) which is number of elements per device, returned by \( \text{cusolverMgSyevd_bufferSize()} \).

\( \text{array_d_A} \) is also a host pointer array of dimension \( G \). \( \text{array_d_A}[j] \) is a device pointer pointing to a device memory in \( j \)-th device. The data type of \( \text{array_d_A}[j] \) is \( \text{dataTypeA} \).
Using the CUSOLVERMG API

The size of array_d_A[j] is about N*TA*(blocks per device). The user has to prepare array_d_A manually (please check the samples in Appendix H).

If output parameter info = -i (less than zero), the i-th parameter is wrong (not counting handle). If info = i (greater than zero), i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If jobz = CUSOLVER_EIG_MODE_VECTOR, A contains the orthonormal eigenvectors of the matrix A. The eigenvectors are computed by a divide and conquer algorithm.

Remark 1: only CUBLAS_FILL_MODE_LOWER is supported, so the user has to prepare lower triangle of A.

Remark 2: only IA=1 and JA=1 are supported.

Remark 3: tile size TA must be less or equal to 1024. To achieve best performance, TA should be 256 or 512.

Appendix H provides three examples of cusolverMgSyevd.

**API of syevd**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: jobz = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; jobz = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>specifies which part of A is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of A is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of A is stored. Only CUBLAS_FILL_MODE_LOWER is supported.</td>
</tr>
<tr>
<td>uplo</td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix sub(A).</td>
</tr>
<tr>
<td>N</td>
<td>host</td>
<td>input</td>
<td>a host pointer array of dimension G. It contains a distributed &lt;type&gt; array containing sub(A) of dimension N * N. If uplo = CUBLAS_FILL_MODE_UPPER, the leading N-by-N upper triangular part of sub(A) contains the upper triangular part of the matrix sub(A). If uplo = CUBLAS_FILL_MODE_LOWER, the leading N-by-N lower triangular part of sub(A) contains the lower triangular part of the matrix sub(A). On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info = 0, sub(A) contains the orthonormal eigenvectors of the matrix sub(A). If jobz = CUSOLVER_EIG_MODE_NOVECTOR, the contents of A are destroyed.</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>IA</td>
<td>host</td>
<td>input</td>
<td>The row index in the global array A indicating the first row of sub(A).</td>
</tr>
<tr>
<td>JA</td>
<td>host</td>
<td>input</td>
<td>The column index in the global array A indicating the first column of sub(A).</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>input</td>
<td>Matrix descriptor for the distributed matrix A.</td>
</tr>
<tr>
<td>W</td>
<td>host</td>
<td>output</td>
<td>A real array of dimension (N). The eigenvalue values of sub(A), in ascending order ie, sorted so that (W(i) \leq W(i+1)).</td>
</tr>
<tr>
<td>dataTypeW</td>
<td>host</td>
<td>input</td>
<td>Data type of the vector W.</td>
</tr>
<tr>
<td>computeType</td>
<td>host</td>
<td>input</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>array_d_work</td>
<td>host</td>
<td>in/out</td>
<td>A host pointer array of dimension G. array_d_work[j] points to a device working space in j-th device, type array of size lwork.</td>
</tr>
<tr>
<td>lwork</td>
<td>host</td>
<td>input</td>
<td>Size of array_d_work[j], returned by cusolverMgSyevd_bufferSize. lwork denotes number of elements, not number of bytes.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>output</td>
<td>If info = 0, the operation is successful. If info = -i, the i-th parameter is wrong (not counting handle). If info = i (&gt; 0), info indicates i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;</td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ((N&lt;0), or (lda&lt;\max(1,N)), or jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or uplo is not CUBLAS_FILL_MODE_LOWER, or IA and JA are not 1, or (N) is bigger than dimension of global A, or the combination of dataType and computeType is not valid.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
Appendix A. cuSolverRF Examples

A.1. cuSolverRF In-memory Example

This is an example in the C programming language of how to use the standard routines in the cuSolverRF library. We focus on solving the set of linear systems

\[ A_i x_i = f_i \]

but we change the indexing from one- to zero-based to follow the C programming language. The example begins with the usual includes and main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cusolverRf.h"

#define TEST_PASSED 0
#define TEST_FAILED 1

int main (void){
    /* matrix A */
    int n;
    int nnzA;
    int *Ap=NULL;
    int *Ai=NULL;
    double *Ax=NULL;
    int *d_Ap=NULL;
    int *d_Ai=NULL;
    double *d_rAx=NULL;
    /* matrices L and U */
    int nnzL, nnzU;
    int *Lp=NULL;
    int *Li=NULL;
    double* Lx=NULL;
    int *Up=NULL;
    int *Ui=NULL;
    double* Ux=NULL;
    /* reordering matrices */
    int *P=NULL;
    int *Q=NULL;
    int * d_P=NULL;
    int * d_Q=NULL;
    /* solution and rhs */
    int nrhs; // # of rhs for each system (currently only =1 is supported)
    double *d_X=NULL;
    double *d_T=NULL;
    /* cuda */
```
cudaError_t cudaStatus;
/* cuolverRf */
cusolverRfHandle_t gH=NULL;
cusolverStatus_t status;
/* host sparse direct solver */
/* ... */
/* other variables */
int tnnzL, tnnzU;
int *tLp=NULL;
int *tLi=NULL;
double *tLx=NULL;
int *tUp=NULL;
int *tUi=NULL;
double *tUx=NULL;
double t1, t2;

Then we initialize the library.

/* ASSUMPTION: recall that we are solving a set of linear systems
A_{i} x_{i} = f_{i}  for i=0,...,k-1
where the sparsity pattern of the coefficient matrices A_{i}
as well as the reordering to minimize fill-in and the pivoting
used during the LU factorization remain the same. */

/* Step 1: solve the first linear system (i=0) on the host,
using host sparse direct solver, which involves
full LU factorization and solve. */
/* ... */

/* Step 2: interface to the library by extracting the following
information from the first solve:
a) triangular factors L and U
b) pivoting and reordering permutations P and Q
c) also, allocate all the necessary memory */
/* ... */

/* Step 3: use the library to solve subsequent (i=1,...,k-1) linear systems
a) the library setup (called only once) */
//create handle
status = cusolverRfCreate(&gH);
if (status != CUSOLVER_STATUS_SUCCESS)&&
    printf ("[cusolverRf status \%d]\n",status);
    return TEST_FAILED;
}

//set fast mode
status = cusolverRfSetResetValuesFastMode(gH,GLU_RESET_VALUES_FAST_MODE_ON);
if (status != CUSOLVER_STATUS_SUCCESS)&&
    printf ("[cusolverRf status \%d]\n",status);
    return TEST_FAILED;
}

Call refactorization and solve.

/* assemble internal data structures (you should use the coefficient matrix A
//corresponding to the second (i=1) linear system in this call)
t1 = cusolver_test_seconds();
status = cusolverRfSetupHost(n, nnzA, Ap, Ai, Ax,
nnzL, Lp, Li, Lx, nnzU, Up, Ui, Ux, P, Q, gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
    printf ("[cusolverRF status \%d]\n",status);
    return TEST_FAILED;
}
printf("cusolverRFSetupHost time = \%f (s)\n", t2-t1);

//analyze available parallelism
t1 = cusolver_test_seconds();
status = cusolverRFAnalyze(gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
    printf ("[cusolverRF status \%d]\n",status);
    return TEST_FAILED;
}
printf("cusolverRFAnalyze time = \%f (s)\n", t2-t1);

/* b) The library subsequent (i=1,...,k-1) LU re-factorization
and solve (called multiple times). */
for (i=1; i<k; i++){
   //LU re-factorization
   t1 = cusolver_test_seconds();
   status = cusolverRFRefactor(gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
    printf ("[cusolverRF status \%d]\n",status);
    return TEST_FAILED;
}
printf("cuSolverReRefactor time = \%f (s)\n", t2-t1);

   //forward and backward solve
   t1 = cusolver_test_seconds();
   status = cusolverRFSolve(gH, d_P, d_Q, nrhs, d_T, n, d_X, n);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
    printf ("[cusolverRF status \%d]\n",status);
    return TEST_FAILED;
}
printf("cusolverRFsSolve time = \%f (s)\n", t2-t1);

// extract the factors (if needed)
status = cusolverRFExtractSplitFactorsHost(gH, &tnnzL, &tLp, &tLi, &tLx,
                                       &tnnzU, &tUp, &tUi, &tUx);
if(status != CUSOLVER_STATUS_SUCCESS){
    printf ("[cusolverRF status \%d]\n",status);
    return TEST_FAILED;
}

/*
 * print
*/
int row, j;
printf("printing L\n");
for (row=0; row<n; row++){
    for (j=tLp[row]; j<tLp[row+1]; j++){
        printf("\%d,\%d,\%f\n",row,tLi[j],tLx[j]);
    }
}
printf("printing U\n");
for (row=0; row<n; row++){
    for (j=tUp[row]; j<tUp[row+1]; j++){
A.2. cuSolverRF-batch Example

This chapter provides an example in the C programming language of how to use the batched routines in the cuSolverRF library. We focus on solving the set of linear systems

\[ A_i x_i = f_i \]

but we change the indexing from one- to zero-based to follow the C programming language. The first part is the usual includes and main definition

```c
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cusolverRf.h"

#define TEST_PASSED   0
#define TEST_FAILED   1

int main (void){
    /* matrix A */
    int batchSize;
    int n;
    int nnzA;
    int *Ap=NULL;
    int *Ai=NULL;
    //array of pointers to the values of each matrix in the batch (of size batchSize) on the host
    double **Ax_array=NULL;
    //For example, if Ax_batch is the array (of size batchSize*nnzA) containing the values of each matrix in the batch written contiguously one matrix
```
Next we initialize the data needed and the create library handles

```c
/* solution and rhs */
int nrhs; // # of rhs for each system (currently only =1 is supported)
// temporary storage (of size 2*batchSize*n*nrhs)
double *d_T=NULL;
// array (of size batchSize*n*nrhs) containing the values of each rhs in
// the batch written contiguously one rhs after another on the device
double **d_X_array=NULL;
// array (of size batchSize*n*nrhs) containing the values of each rhs in
// the batch written contiguously one rhs after another on the host
double ***X_array=NULL;
/* cuda */
cudaError_t cudaStatus;
/* cusolverRf */
cusolverRfHandle_t gH=NULL;
cusolverStatus_t status;
/* host sparse direct solver */
...
/* other variables */
double t1, t2;

/* ASSUMPTION:
 recall that we are solving a batch of linear systems
 A_{j} x_{j} = f_{j} for j=0,...,batchSize-1
 where the sparsity pattern of the coefficient matrices A_{j} as well as the
 reordering to minimize fill-in and the pivoting used during the LU
 factorization remain the same. */

/* Step 1: solve the first linear system (j=0) on the host,
 using host sparse direct solver, which involves
 full LU factorization and solve. */
/* ... */

/* Step 2: interface to the library by extracting the following
 information from the first solve:
 a) triangular factors L and U 
```
b) pivoting and reordering permutations \( P \) and \( Q \)
c) also, allocate all the necessary memory */

/* Step 3: use the library to solve the remaining \((j=1,\ldots,\text{batchSize}-1)\) linear systems.
a) the library setup (called only once) */

//create handle
status = cusolverRfcreate(&gH);
if (status != CUSOLVER_STATUS_SUCCESS){
  printf ("[cusolverRf status %d]\\n",status);
  return TEST_FAILED;
}

We call the batch solve method and return.

//assemble internal data structures
t1 = cusolver_test_seconds();
status = cusolverRfBatchSetupHost(batchSize, n, nnzA, Ap, Ai, Ax_array,
  nnzL, Lp, Li, Lx, nnzU, Up, Ui, Ux, P, Q, gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
  printf ("[cusolverRf status %d]\\n",status);
  return TEST_FAILED;
}
printf("cusolverRfBatchSetupHost time = %f (s)\\n", t2-t1);

//analyze available parallelism
t1 = cusolver_test_seconds();
status = cusolverRfBatchAnalyze(gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
  printf ("[cusolverRf status %d]\\n",status);
  return TEST_FAILED;
}
printf("cusolverRfBatchAnalyze time = %f (s)\\n", t2-t1);

/* b) The library subsequent \((j=1,\ldots,\text{batchSize}-1)\) LU re-factorization
and solve (may be called multiple times). For the subsequent batches
the values can be reset using cusolverRfBatch_reset_values_routine. */

//LU re-factorization
t1 = cusolver_test_seconds();
status = cusolverRfBatchRefactor(gH);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
  printf ("[cusolverRf status %d]\\n",status);
  return TEST_FAILED;
}
printf("cusolverRfBatchRefactor time = %f (s)\\n", t2-t1);

//forward and backward solve
t1 = cusolver_test_seconds();
status = cusolverRfBatchSolve(gH, d_P, d_Q, nrhs, d_T, n, d_X_array, n);
cudaStatus = cudaDeviceSynchronize();
t2 = cusolver_test_seconds();
if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
  printf ("[cusolverRf status %d]\\n",status);
  return TEST_FAILED;
}
printf("cusolverRfBatchSolve time = %f (s)\\n", t2-t1);

/* free memory and exit */
return TEST_PASSED;
}
Appendix B. CSR QR Batch Examples

B.1. Batched Sparse QR example 1

This chapter provides a simple example in the C programming language of how to use batched sparse QR to solve a set of linear systems

\[ A_i x_i = b_i \]

All matrices \( A_i \) are small perturbations of

\[
A = \begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 0.0 \\
0.1 & 0.1 & 0.1 & 4.0
\end{bmatrix}
\]

All right-hand side vectors \( b_i \) are small perturbation of the Matlab vector ‘ones(4,1)’.

We assume device memory is big enough to compute all matrices in one pass.

The usual includes and main definition

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cusolverSp.h>
#include <cuda_runtime_api.h>

int main(int argc, char*argv[])
{
    cusolverSpHandle_t cusolverH = NULL;
    // GPU does batch QR
    csrqrInfo_t info = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseStatus_t cusparse_status = CUSPARSE_STATUS_SUCCESS;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    // GPU does batch QR
    // d_A is CSR format, d_csrValA is of size nnzA*batchSize
    // d_x is a matrix of size batchSize * m
```
Set up the library handle and data

```c
const int m = 4;
const int nnzA = 7;
const int csrRowPtrA[m+1] = { 1, 2, 3, 4, 8};
const int csrColIndA[nnzA] = { 1, 2, 3, 1, 2, 3, 4};
const double csrValA[nnzA] = { 1.0, 2.0, 3.0, 0.1, 0.1, 0.1, 4.0};
const double b[m] = {1.0, 1.0, 1.0, 1.0};
const int batchSize = 17;

double *csrValABatch = (double*)malloc(sizeof(double)*nnzA*batchSize);
double *bBatch       = (double*)malloc(sizeof(double)*m*batchSize);
double *xBatch       = (double*)malloc(sizeof(double)*m*batchSize);
assert( NULL != csrValABatch );
assert( NULL != bBatch );
assert( NULL != xBatch );

// step 1: prepare Aj and bj on host
// Aj is a small perturbation of A
// bj is a small perturbation of b
// csrValABatch = [A0, A1, A2, ...]
// bBatch = [b0, b1, b2, ...]
for(int colidx = 0 ; colidx < nnzA ; colidx++){
    double Areg = csrValA[colidx];
    for (int batchId = 0 ; batchId < batchSize ; batchId++){
        double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
        csrValABatch[batchId*nnzA + colidx] = Areg + eps;
    }
}
for(int j = 0 ; j < m ; j++){
    double breg = b[j];
    for (int batchId = 0 ; batchId < batchSize ; batchId++){
        double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
        bBatch[batchId*m + j] = breg + eps;
    }
}

// step 2: create cusolver handle, qr info and matrix descriptor
 cusolver_status = cusolverSpCreate(&cusolverH);
 assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cusparse_status = cusparseCreateMatDescr(&descrA);
assert(cusparse_status == CUSPARSE_STATUS_SUCCESS);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
```

cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);  // base-1

cusolver_status = cusolverSpCreateCsrqrInfo(&info);
assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

Call the solver

// step 3: copy Aj and bj to device
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(double) * nnzA * batchSize);
    cudaStat2 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int) * nnzA);
    cudaStat3 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int) * (m+1));
    cudaStat4 = cudaMalloc ((void**)&d_b         , sizeof(double) * m * batchSize);
    cudaStat5 = cudaMalloc ((void**)&d_x         , sizeof(double) * m * batchSize);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);
    assert(cudaStat4 == cudaSuccess);
    assert(cudaStat5 == cudaSuccess);

    cudaMemcpy(d_csrValA   , csrValABatch, sizeof(double) * nnzA * batchSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, bBatch, sizeof(double) * m * batchSize, cudaMemcpyHostToDevice);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);
    assert(cudaStat4 == cudaSuccess);

    cusolver_status = cusolverSpXcsrqrAnalysisBatched(
        cusolverH, m, m, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        info);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

    cusolver_status = cusolverSpDcsrqrBufferInfoBatched(
        cusolverH, m, m, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        batchSize,
        info,
        &size_internal,
        &size_qr);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

    printf("numerical factorization needs internal data %lld bytes\n",
           (long long)size_internal);
    printf("numerical factorization needs working space %lld bytes\n",
           (long long)size_qr);

    cudaStat1 = cudaMalloc((void**)&buffer_qr, size_qr);
    assert(cudaStat1 == cudaSuccess);

Get results back

// step 4: symbolic analysis
    cusolver_status = cusolverSpXcsrqrAnalysisBatched(
        cusolverH, m, m, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        info);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

    cusolver_status = cusolverSpDcsrqrBufferInfoBatched(
        cusolverH, m, m, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        batchSize,
        info,
        &size_internal,
        &size_qr);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

    printf("numerical factorization needs internal data %lld bytes\n",
           (long long)size_internal);
    printf("numerical factorization needs working space %lld bytes\n",
           (long long)size_qr);

    cudaStat1 = cudaMalloc((void**)&buffer_qr, size_qr);
    assert(cudaStat1 == cudaSuccess);

// step 6: numerical factorization
// assume device memory is big enough to compute all matrices.
    cusolver_status = cusolverSpDcsrqrsvBatched(
        cusolverH, m, m, nnzA,
B.2. Batched Sparse QR example 2

This is the same as example 1 in appendix C except that we assume device memory is not enough, so we need to cut 17 matrices into several chunks and compute each chunk by batched sparse QR.

The usual includes and main definitions

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cusolverSp.h>
```
#include <cuda_runtime_api.h>

#define imin( x, y ) ((x)<(y))? (x) : (y)

int main(int argc, char*argv[])
{
    cusolverSpHandle_t cusolverH = NULL;
    // GPU does batch QR
    csrqrInfo_t info = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseStatus_t cusparse_status = CUSPARSE_STATUS_SUCCESS;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    // GPU does batch QR
    // d_A is CSR format, d_csrValA is of size nnzA*batchSize
    // d_x is a matrix of size batchSize * m
    // d_b is a matrix of size batchSize * m
    int *d_csrRowPtrA = NULL;
    int *d_csrColIndA = NULL;
    double *d_csrValA = NULL;
    double *d_b = NULL; // batchSize * m
    double *d_x = NULL; // batchSize * m
    size_t size_qr = 0;
    size_t size_internal = 0;
    void *buffer_qr = NULL; // working space for numerical factorization
    /*
     *  A = |
     *  | 1  |
     *  | 2  |
     *  | 3  |
     *  | 0.1 0.1 0.1 4 |
     * CSR of A is based-1
     * b = [1 1 1 1]
     */

    const int m = 4 ;
    const int nnzA = 7;
    const int csrRowPtrA[m+1]  = { 1, 2, 3, 4, 8};
    const int csrColIndA[nnzA] = { 1, 2, 3, 1, 2, 3, 4};
    const double csrValA[nnzA] = { 1.0, 2.0, 3.0, 0.1, 0.1, 0.1, 4.0};
    const double b[m] = {1.0, 1.0, 1.0, 1.0};
    const int batchSize = 17;
    double *csrValABatch = (double*)malloc(sizeof(double)*nnzA*batchSize);
    double *bBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    double *xBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    assert( NULL != csrValABatch );
    assert( NULL != bBatch );
    assert( NULL != xBatch );

    // step 1: prepare Aj and bj on host
    // Aj is a small perturbation of A
    // bj is a small perturbation of b
    // csrValABatch = [A0, A1, A2, ...]
    // bBatch = [b0, b1, b2, ...]
    for(int colidx = 0 ; colidx < nnzA ; colidx++)
    {
        double Areg = csrValA[colidx];
        ...
for (int batchId = 0; batchId < batchSize; batchId++){
    double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
    csrValABatch[batchId*nnzA + colidx] = Areg + eps;
}

for(int j = 0; j < m; j++){
    double breg = b[j];
    for (int batchId = 0; batchId < batchSize; batchId++){
        double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
        bBatch[batchId*m + j] = breg + eps;
    }
}

// step 2: create cusolver handle, qr info and matrix descriptor
cusolver_status = cusolverSpCreate(&cusolverH);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cusparse_status = cusparseCreateMatDescr(&descrA);
assert(cusparse_status == CUSPARSE_STATUS_SUCCESS);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE); // base-1

cusolver_status = cusolverSpCreateCsrqrInfo(&info);
assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

Set up the data

// step 3: copy Aj and bj to device
cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(double) * nnzA * batchSize);
cudaStat2 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int) * nnzA);
cudaStat3 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int) * (m+1));
cudaStat4 = cudaMalloc ((void**)&d_b         , sizeof(double) * m * batchSize);
cudaStat5 = cudaMalloc ((void**)&d_x         , sizeof(double) * m * batchSize);
assert(cudaStat1 == cudaSuccess);
assert(cudaStat2 == cudaSuccess);
assert(cudaStat3 == cudaSuccess);
assert(cudaStat4 == cudaSuccess);
assert(cudaStat5 == cudaSuccess);

// don't copy csrValABatch and bBatch because device memory may be big enough
cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA,
cudaStat2 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1),
cudaStat1 == cudaSuccess);
assert(cudaStat2 == cudaSuccess);
assert(cudaStat1 == cudaSuccess);
assert(cudaStat2 == cudaSuccess);

// step 4: symbolic analysis

// step 5: find "proper" batchSize
// get available device memory
size_t free_mem = 0;
size_t total_mem = 0;
cudaStat1 = cudaMemGetInfo( &free_mem, &total_mem );
assert( cudaSuccess == cudaStat1 );

int batchSizeMax = 2;
while(batchSizeMax < batchSize){
    printf("batchSizeMax = %d\n", batchSizeMax);
    cusolver_status = cusolverSpDcsrqrBufferInfoBatched(
        cusolverH, m, m, nnzA,
        // d_csrValA is don't care
descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
        batchSizeMax, // WARNING: use batchSizeMax
        info,
        &size_internal,
        &size_qr);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);
    if ( (size_internal + size_qr) > free_mem ){
        // current batchSizeMax exceeds hardware limit, so cut it by half.
        batchSizeMax /= 2; break;
    } else {
        batchSizeMax *= 2; // double batchSizeMax and try it again.
    }
}
// correct batchSizeMax such that it is not greater than batchSize.
batchSizeMax = imin(batchSizeMax, batchSize);
printf("batchSizeMax = %d\n", batchSizeMax);
if (batchSizeMax == 2)
    batchSizeMax = 2;

Perform analysis and call solve

// step 6: prepare working space
// [necessary]
// Need to call cusolverDcsrqrBufferInfoBatched again with batchSizeMax
// to fix batchSize used in numerical factorization.
cusolver_status = cusolverSpDcsrqrBufferInfoBatched(
    cusolverH, m, m, nnzA,
    // d_csrValA is don't care
descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
    batchSizeMax, // WARNING: use batchSizeMax
    info,
    &size_internal,
    &size_qr);
assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);
printf("numerical factorization needs internal data %lld bytes\n",
    (long long)size_internal);
printf("numerical factorization needs working space %lld bytes\n",
    (long long)size_qr);
    cudaStat1 = cudaMalloc((void**)&buffer_qr, size_qr);
    assert(cudaStat1 == cudaSuccess);
    printf("buffer_qr allocated\n");
// step 7: solve Aj*xj = bj
for(int idx = 0 ; idx < batchSize; idx += batchSizeMax){
    printf("current batchSize 'cur_batchSize' is the batchSize used in numerical factorization\n");
    // copy part of Aj and bj to d
    cudaMemcpy(d_csrValA, csrValABatch + idx*nnzA,
        sizeof(double) * nnzA * cur_batchSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, bBatch + idx*m,
        sizeof(double) * m * cur_batchSize, cudaMemcpyHostToDevice);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    // solve part of Aj*xj = bj
    cusolver_status = cusolverSpDcsrqrsvBatched(
        cusolverH, m, m, nnzA,
Check results

// step 7: check residual
// xBatch = [x0, x1, x2, ...]
const int baseA = (CUSPARSE_INDEX_BASE_ONE == cusparseGetMatIndexBase descrA))? 1:0 ;

for(int batchId = 0 ; batchId < batchSize; batchId++){
    // measure |bj - Aj*xj|
    double *csrValAj = csrValABatch + batchId * nnzA;
    double *xj = xBatch + batchId * m;
    double *bj = bBatch + batchId * m;
    // sup| bj - Aj*xj|
    double sup_res = 0;
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row] - baseA;
        const int end = csrRowPtrA[row+1] - baseA;
        double Ax = 0.0; // Aj(row,:)*xj
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - baseA;
            const double Areg = csrValAj[colidx];
            const double xreg = xj[col];
            Ax = Ax + Areg * xreg;
        }
        double r = bj[row] - Ax;
        sup_res = (sup_res > fabs(r))? sup_res : fabs(r);
    }
    printf("batchId %d: sup|bj - Aj*xj| = %E \n", batchId, sup_res);
}

for(int batchId = 0 ; batchId < batchSize; batchId++){
    double *xj = xBatch + batchId * m;
    for(int row = 0 ; row < m ; row++){
        printf("x%d[%d] = %E\n", batchId, row, xj[row]);
    }
    printf("\n");
}

return 0;
Appendix C. QR Examples

C.1. QR Factorization Dense Linear Solver

This chapter provides a simple example in the C programming language of how to use a dense QR factorization to solve a linear system

\[ Ax = b \]

A is a 3x3 dense matrix, nonsingular.

\[
A = \begin{pmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
2.0 & 1.0 & 1.0
\end{pmatrix}
\]

The following code uses three steps:

Step 1: \( A = QR \) by \text{geqrf}.

Step 2: \( B := Q^T B \) by \text{ormqr}.

Step 3: solve \( R^X = B \) by \text{trsm}.

The usual includes and main definition

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name) {
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
```
```c
int main(int argc, char*argv[])
{

cusolverDnHandle_t cusolverH = NULL;
cublasHandle_t cublasH = NULL;
cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;
cudaError_t cudaStat2 = cudaSuccess;
cudaError_t cudaStat3 = cudaSuccess;
cudaError_t cudaStat4 = cudaSuccess;
const int m = 3;
const int lda = m;
const int ldb = m;
const int nrhs = 1; // number of right hand side vectors

/*
 | 1 2 3 |
* A = | 4 5 6 |
* | 2 1 1 |
* x = (1 1 1)'
* b = (6 15 4)'
*/

Create the library handle and load the data

```
```c
// Query working space of geqrf and ormqr

// Query working space of geqrf

cusolverStatus_t cusolverStatus = cusolverDnDgeqrf_bufferSize(
    cudenseH,
    m,
    d_A,
    lda,
    &lwork_geqrf);
assert (cusolverStatus == CUSOLVER_STATUS_SUCCESS);

// Query working space of ormqr

cusolverStatus = cusolverDnDormqr_bufferSize(
    cudenseH,
    CUBLAS_SIDE_LEFT,
    CUBLAS_OP_T,
    m,
    nrhs,
    m,
    d_A,
    lda,
    d_tau,
    d_B,
    ldb,
    &lwork_ormqr);
assert (cusolverStatus == CUSOLVER_STATUS_SUCCESS);

lwork = (lwork_geqrf > lwork_ormqr) ? lwork_geqrf : lwork_ormqr;

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m , cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * ldb * nrhs, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

/* step 4: compute QR factorization */
cusolverStatus = cusolverDnDgeqrf(
    cusolverH,
    m,
    m,
    d_A,
    lda,
    d_tau,
    d_work,
    lwork,
    devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);

/* check if QR is good or not */
cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
```

Call the solver

```c
/* step 3: query working space of geqrf and ormqr */
cusolver_status = cusolverDnDgeqrf_bufferSize(
    cudenseH,
    m,
    m,
    d_A,
    lda,
    &lwork_geqrf);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cusolver_status= cusolverDnDormqr_bufferSize(
    cudenseH,
    CUBLAS_SIDE_LEFT,
    CUBLAS_OP_T,
    m,
    nrhs,
    m,
    d_A,
    lda,
    d_tau,
    d_B,
    ldb,
    &lwork_ormqr);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

lwork = (lwork_geqrf > lwork_ormqr) ? lwork_geqrf : lwork_ormqr;

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m , cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * ldb * nrhs, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
```

Call the solver

```c
/* step 3: query working space of geqrf */
cusolver_status = cusolverDnDgeqrf_bufferSize(
    cudenseH,
    m,
    m,
    d_A,
    lda,
    &lwork_geqrf);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cusolver_status= cusolverDnDormqr_bufferSize(
    cudenseH,
    CUBLAS_SIDE_LEFT,
    CUBLAS_OP_T,
    m,
    nrhs,
    m,
    d_A,
    lda,
    d_tau,
    d_B,
    ldb,
    &lwork_ormqr);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

lwork = (lwork_geqrf > lwork_ormqr) ? lwork_geqrf : lwork_ormqr;

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m , cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * ldb * nrhs, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
```
Check the results

```c
/* step 5: compute Q^T*B */
cusolver_status = cusolverDnDormqr(
    cudenseH,
    CUBLAS_SIDE_LEFT,
    CUBLAS_OP_T,
    m,
    nrhs,
    m,
    d_A,
    lda,
    d_tau,
    d_B,
    ldb,
    d_work,
    lwork,
    devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);
/* check if QR is good or not */
cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printf("after ormqr: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);
/* step 6: compute x = R \ Q^T*B */
cublas_status = cublasDtrsm(
    cublasH,
    CUBLAS_SIDE_LEFT,
    CUBLAS_FILL_MODE_UPPER,
    CUBLAS_OP_N,
    CUBLAS_DIAG_NON_UNIT,
    m,
    nrhs,
    &one,
    d_A,
    lda,
    d_B,
    ldb);
cudaStat1 = cudaDeviceSynchronize();
assert(CUBLAS_STATUS_SUCCESS == cublas_status);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(XC, d_B, sizeof(double)*ldb*nrhs,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printf("X = (matlab base-1)\n");
printMatrix(m, nrhs, XC, ldb, "X");
/* free resources */
if (d_A    ) cudaFree(d_A);
if (d_tau  ) cudaFree(d_tau);
if (d_B    ) cudaFree(d_B);
if (devInfo) cudaFree(devInfo);
if (d_work ) cudaFree(d_work);
if (cublasH ) cublasDestroy(cublasH);
if (cusolverH) cusolverDnDestroy(cusolverH);
cudaDeviceReset();
```
C.2. orthogonalization

This chapter provides a simple example in the C programming language of how to do orthogonalization by QR factorization.

\[ A = Q^*R \]

A is a 3x2 dense matrix,

\[
A = \begin{bmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0
\end{bmatrix}
\]

The following code uses three steps:

Step 1: \( A = Q^*R \) by \texttt{geqrf}.

Step 2: form \( Q \) by \texttt{orgqr}.

Step 3: check if \( Q \) is unitary or not.

The usual includes and main definition

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cublasHandle_t cublasH = NULL;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    ...
Create the library handle and load the data

```c
double A[lda*n] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0};
double Q[lda*n]; // orthonormal columns
double R[n*n]; // R = I - Q**T*Q

double *d_A = NULL;
double *d_tau = NULL;
int *devInfo = NULL;
double *d_work = NULL;
double *d_R = NULL;
int lwork_geqrf = 0;
int lwork_orgqr = 0;
int lwork = 0;
int info_gpu = 0;
const double h_one = 1;
const double h_minus_one = -1;
printf("A = (matlab base-1)\n");
printMatrix(m, n, A, lda, "A");
printf("=====
");

// step 1: create cusolverDn/cublas handle
cusolver_status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
cublas_status = cublasCreate(&cublasH);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

// step 2: copy A and B to device
cudaStat1 = cudaMalloc ((void**)&d_A  , sizeof(double)*lda*n);
cudaStat2 = cudaMalloc ((void**)&d_tau, sizeof(double)*n);
cudaStat3 = cudaMalloc ((void**)&devInfo, sizeof(int));
cudaStat4 = cudaMalloc ((void**)&d_R  , sizeof(double)*n*n);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
```

Call the solver

```c
// step 3: query working space of geqrf and orgqr
cusolver_status = cusolverDnDgeqrf_bufferSize(
    cusolverH,
```
m,
n,
d_A,
lda,
&lwork_geqrf);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);
cusolver_status = cusolverDnDorgqr_bufferSize(
cusolverH,
m,
n,
n,
d_A,
lda,
&lwork_orgqr);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);
// lwork = max(lwork_geqrf, lwork_orgqr)
lwork = (lwork_geqrf > lwork_orgqr) ? lwork_geqrf : lwork_orgqr;
cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

// step 4: compute QR factorization
cusolver_status = cusolverDnDgeqrf(
cusolverH,
m,
d_A,
lda,
d_tau,
d_work,
lwork,
devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);

// check if QR is successful or not
cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printf("after geqrf: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);

// step 5: compute Q
cusolver_status= cusolverDnDorgqr(
cusolverH,
m,
n,
d_A,
lda,
d_tau,
d_work,
lwork,
devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);

Check the results

// check if QR is good or not
cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
```c
printf("after orgqr: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);

cudaStat1 = cudaMemcpy(Q, d_A, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);

printf("Q = (matlab base-1)\n");
printMatrix(m, n, Q, lda, "Q");

// step 6: measure R = I - Q**T*Q
memset(R, 0, sizeof(double)*n*n);
for(int j = 0 ; j < n ; j++){
    R[j + n*j] = 1.0; // R(j,j)=1
}

cudaStat1 = cudaMemcpy(d_R, R, sizeof(double)*n*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

// R = -Q**T*Q + I

double dR_nrm2 = 0.0;
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

cublasStatus = cublasDnrm2_v2(
    &h_minus_one, /* host pointer */
    d_A, // Q**T
    lda,
    d_A, // Q
    lda,
    &h_one, /* host pointer */
    d_R,
    n);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

double dR_nrm2 = 0.0;
cublasStatus = cublasDnrm2_v2(
    &h_one, /* host pointer */
    d_R,
    n);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

printf("|I - Q**T*Q| = %E\n", dR_nrm2);

free resources

// free resources
if (d_A   ) cudaFree(d_A);
if (d_tau  ) cudaFree(d_tau);
if (devInfo) cudaFree(devInfo);
if (d_work ) cudaFree(d_work);
if (d_R    ) cudaFree(d_R);
if (cublasH ) cublasDestroy(cublasH);
if (cusolverH) cusolverDnDestroy(cusolverH);

cudaDeviceReset();

return 0;
```
Appendix D. LU Examples

D.1. LU Factorization

This chapter provides a simple example in the C programming language of how to use a dense LU factorization to solve a linear system

\[ Ax = b \]

A is a 3x3 dense matrix, nonsingular.

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
7.0 & 8.0 & 10.0 \\
\end{bmatrix}
\]

The code uses getrf to do LU factorization and getrs to do backward and forward solve. The parameter pivot_on decides whether partial pivoting is performed or not.

...
double A[lda*m] = { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 10.0};
double B[m] = { 1.0, 2.0, 3.0 };
double X[m]; /* X = A\B */
double LU[lda*m]; /* L and U */
int Ipiv[m]; /* host copy of pivoting sequence */
int info = 0; /* host copy of error info */

double *d_A = NULL; /* device copy of A */
double *d_B = NULL; /* device copy of B */
int *d_Ipiv = NULL; /* pivoting sequence */
int *d_info = NULL; /* error info */
int lwork = 0; /* size of workspace */
double *d_work = NULL; /* device workspace for getrf */

const int pivot_on = 0;
printf("example of getrf \n");
if (pivot_on){
    printf("pivot is on : compute P*A = L*U \n");
} else{
    printf("pivot is off: compute A = L*U (not numerically stable)\n");
}

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("====\n");

printf("B = (matlab base-1)\n");
printMatrix(m, l, B, ldb, "B");
printf("====\n");

/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);
```c
CUDAStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == CUDAStat1);

status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: copy A to device */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
cudaStat2 = cudaMalloc ((void**)&d_B, sizeof(double) * m);
cudaStat2 = cudaMalloc ((void**)&d_Ipiv, sizeof(int) * m);
cudaStat4 = cudaMalloc ((void**)&d_info, sizeof(int));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*m, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double)*m, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
...

/* step 3: query working space of getrf */
status = cusolverDnDgetrf_bufferSize(cusolverH, m, m, d_A, lda, &lwork);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

/* step 4: LU factorization */
if (pivot_on){
  status = cusolverDnDgetrf(cusolverH, m, m, d_A, lda, d_work, d_Ipiv, d_info);
} else{
  status = cusolverDnDgetrf(cusolverH, m, m, d_A, lda, d_work, NULL, NULL, d_info);
}

cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);
if (pivot_on){
  cudaStat1 = cudaMemcpy(Ipiv, d_Ipiv, sizeof(int)*m, cudaMemcpyDeviceToHost);
}
```
cudaStat2 = cudaMemcpy(LU , d_A , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(&info, d_info, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

if ( 0 > info )
   printf("%d-th parameter is wrong \n", -info);
   exit(1);
} 
if (pivot_on)
   printf("pivoting sequence, matlab base-1\n");
   for(int j = 0 ; j < m ; j++)
      printf("Ipiv(%d) = %d\n", j+1, Ipiv[j]);
}
printf("L and U = (matlab base-1)\n");
printMatrix(m, m, LU, lda, "LU");
printf("====\n");

/*
* step 5: solve A*X = B
*   | 1 |       | -0.3333 |
*   | 2 |,  X = |  0.6667 |
*   | 3 |       |  0      |
* 
*/
if (pivot_on){
   status = cusolverDnDgetrs(
      cusolverH,
      CUBLAS_OP_N,
      m,
      1, /* nrhs */
      d_A,
      lda,
      d_Ipiv,
      d_B,
      ldb,
      d_info);
}else{
   status = cusolverDnDgetrs(
      cusolverH,
      CUBLAS_OP_N,
      m,
      1, /* nrhs */
      d_A,
      lda,
      NULL,
      d_B,
      ldb,
      d_info);
}
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(X , d_B, sizeof(double)*m, cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printf("X = (matlab base-1)\n");
printMatrix(m, 1, X, ldb, "X");
printf("====\n");
/* free resources */
    if (d_A   ) cudaFree(d_A);
    if (d_B   ) cudaFree(d_B);
    if (d_Ipiv) cudaFree(d_Ipiv);
    if (d_info) cudaFree(d_info);
    if (d_work) cudaFree(d_work);

    if (cusolverH ) cusolverDnDestroy(cusolverH);
    if (stream     ) cudaStreamDestroy(stream);

cudaDeviceReset();
    return 0;
}
Appendix E. Cholesky Examples

E.1. batched Cholesky Factorization

This chapter provides a simple example in the C programming language of how to use a batched dense Cholesky factorization to solve a sequence of linear systems

\[ A[i]x[i] = b[i] \]

each \( A[i] \) is a 3x3 dense Hermitian matrix. In this example, there are two matrices, \( A_0 \) and \( A_1 \). \( A_0 \) is positive definite and \( A_1 \) is not.

The code uses \texttt{potrfBatched} to do Cholesky factorization and \texttt{potrsBatched} to do backward and forward solve. \texttt{potrfBatched} would report singularity on \( A_1 \).

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++)
    {
        for(int col = 0 ; col < n ; col++)
        {
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t handle = NULL;
    cudaStream_t stream = NULL;

    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
```
double A0[lda*m] = { 1.0, 2.0, 3.0, 2.0, 5.0, 5.0, 3.0, 5.0, 12.0 };
double A1[lda*m] = { 1.0, 2.0, 3.0, 2.0, 4.0, 5.0, 3.0, 5.0, 12.0 };
double B0[m] = { 1.0, 1.0, 1.0 };
double X0[m]; /* X0 = A0\B0 */
int infoArray[batchSize]; /* host copy of error info */
double L0[lda*m]; /* cholesky factor of A0 */

double *Aarray[batchSize];
double *Barray[batchSize];
double **d_Aarray = NULL;
double **d_Barray = NULL;
int *d_infoArray = NULL;

printf("example of batched Cholesky \n");
printf("A0 = (matlab base-1)\n");
printfMatrix(m, m, A0, lda, "A0");
printf("=====
");
printf("A1 = (matlab base-1)\n");
printfMatrix(m, m, A1, lda, "A1");
printf("=====
");
printf("B0 = (matlab base-1)\n");
printfMatrix(m, 1, B0, ldb, "B0");
printf("=====
");

/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusolverDnSetStream(handle, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: copy A to device */
for(int j = 0 ; j < batchSize ; j++){
    cudaStat1 = cudaMalloc ((void**) &Aarray[j], sizeof(double) * lda * m);
    assert(cudaSuccess == cudaStat1);
    cudaStat2 = cudaMalloc ((void**) &Barray[j], sizeof(double) * ldb * nrhs);
    assert(cudaSuccess == cudaStat2);
}

cudaStat1 = cudaMalloc ((void**) &d_infoArray, sizeof(int)*batchSize);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(Aarray[0], A0, sizeof(double) * lda * m,
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(Barray[0], B0, sizeof(double) * m,
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_Aarray, Aarray, sizeof(double*)*batchSize,
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpyp(Aarray[0], A0, sizeof(double) * lda * m, 
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_Barray, Barray, sizeof(double*)*batchSize,
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(Barray[0], B0, sizeof(double) * m, 
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_Barray, Barray, sizeof(double*)*batchSize,
            cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaDeviceSynchronize();

/* step 3: Cholesky factorization */
status = cusolverDnDpotrfBatched(
    handle,
    uplo,
    m,
    d_Aarray,
    lda,
    d_infoArray,
    batchSize);
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(infoArray, d_infoArray, sizeof(int)*batchSize,
            cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(L0, Aarray[0], sizeof(double) * lda * m,
            cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);

cudaDeviceSynchronize();

for(int j = 0 ; j < batchSize ; j++){
    printf("info[%d] = %d\n", j, infoArray[j]);
}

assert( 0 == infoArray[0] );
/* A1 is singular */
assert( 2 == infoArray[1] );

printf("L = (matlab base-1), upper triangle is don't care \n");
printMatrix(m, m, L0, lda, "L0");
printf("====\n");

/*
 * step 4: solve A0*X0 = B0
 * | 1 |        | 10.5 |
 * B0 = | 1 |,  X0 = | -2.5 |
 * | 1 |        | -1.5 |
 */
status = cusolverDnDpotrsBatched(
    handle,
    uplo,
    m,
    nrhs, /* only support rhs = 1*/
    d_Aarray,
    lda,
    d_Barray,
    ldb,
    d_infoArray,
    batchSize);

...
Appendix F. Examples of Dense Eigenvalue Solver

F.1. Standard Symmetric Dense Eigenvalue Solver

This chapter provides a simple example in the C programming language of how to use syevd to compute the spectrum of a dense symmetric system by

$$Ax = \lambda x$$

where $A$ is a 3x3 dense symmetric matrix

$$A = \begin{pmatrix} 3.5 & 0.5 & 0 \\ 0.5 & 3.5 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

The following code uses syevd to compute eigenvalues and eigenvectors, then compare to exact eigenvalues \{2,3,4\}.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name) {
    for(int row = 0 ; row < m ; row++) {
        for(int col = 0 ; col < n ; col++) { double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}
```

/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include syevd_example.cpp
 *   g++ -o a.out syevd_example.o -L/usr/local/cuda/lib64 -lcudart -lcusolver
 * *
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name) {
    for(int row = 0 ; row < m ; row++) {
        for(int col = 0 ; col < n ; col++) { double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}
int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    const int m = 3;
    const int lda = m;
    /*
    * A = | 3.5 0.5 0 |
    *    | 0.5 3.5 0 |
    *    | 0   0   2 |
    */
    double A[lda*m] = { 3.5, 0.5, 0, 0.5, 3.5, 0, 0, 0, 2.0};
    double lambda[m] = { 2.0, 3.0, 4.0};
    double V[lda*m]; // eigenvectors
    double W[m]; // eigenvalues
    double *d_A = NULL;
    double *d_W = NULL;
    int *devInfo = NULL;
    double *d_work = NULL;
    int lwork = 0;
    int info_gpu = 0;
    printf("A = (matlab base-1)\n");
    printMatrix(m, m, A, lda, "A");
    printf("=====
");
    call eigenvalue solver
    // step 1: create cusolver/cublas handle
    cusolver_status = cusolverDnCreate(&cusolverH);
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
    // step 2: copy A and B to device
    cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
    cudaStat2 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
    cudaStat3 = cudaMalloc ((void**)&devInfo, sizeof(int));
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
    cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m,
                            cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    // step 3: query working space of syevd
    cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvalues and
    eigenvectors.
    cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
    cusolver_status = cusolverDnDsyevd_bufferSize{
        cusolverH,
        jobz,
        uplo,
        m,
        d_A,
        lda,
        d_W,
        &lwork};
    assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);
cudastat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudastat1);

// step 4: compute spectrum
 cusolver_status = cusolverDnDsyevd(
 cusolverH,
 jobz,
 uplo,
 m,
 d_A,
 lda,
 d_W,
 d_work,
 lwork,
 devInfo);
 cudastat1 = cudaDeviceSynchronize();
 assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
 assert(cudaSuccess == cudastat1);

cudastat1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudastat2 = cudaMemcpy(A, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudastat3 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudastat1);
assert(cudaSuccess == cudastat2);
assert(cudaSuccess == cudastat3);

check the result

printf("after syevd: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);

printf("eigenvalue = (matlab base-1), ascending order\n");
for(int i = 0 ; i < m ; i++){
    printf("W[%d] = %E\n", i+1, W[i]);
}

printf("V = (matlab base-1)\n");
printMatrix(m, m, V, lda, "V");
printf("====\n");

// step 4: check eigenvalues
 double lambda_sup = 0;
 for(int i = 0 ; i < m ; i++){
    double error = fabs( lambda[i] - W[i]);
    lambda_sup = (lambda_sup > error)? lambda_sup : error;
 }
 printf("|lambda - W| = %E\n", lambda_sup);

// free resources
 if (d_A    ) cudaFree(d_A);
 if (d_W    ) cudaFree(d_W);
 if (devInfo) cudaFree(devInfo);
 if (d_work ) cudaFree(d_work);

 if (cusolverH) cusolverDnDestroy(cusolverH);
 cudaDeviceReset();
 return 0;
F.2. Standard Symmetric Dense Eigenvalue Solver

This chapter provides a simple example in the C programming language of how to use syevd to compute the spectrum of a dense symmetric system by

\[ Ax = \lambda x \]

where \( A \) is a 3x3 dense symmetric matrix

\[
A = \begin{pmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2
\end{pmatrix}
\]

The following code uses syevd to compute eigenvalues and eigenvectors, then compare to exact eigenvalues \([2, 3, 4]\).

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    const int m = 3;
    const int lda = m;
    double A[lda*m] = { 3.5, 0.5, 0, 0.5, 3.5, 0, 0, 0, 2.0};
    double lambda[m] = { 2.0, 3.0, 4.0};
    double V[lda*m]; // eigenvectors
```
double W[m]; // eigenvalues

double *d_A = NULL;
double *d_W = NULL;
int *devInfo = NULL;
double *d_work = NULL;
int lwork = 0;

int info_gpu = 0;

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("==\n");
call eigenvalue solver

// step 1: create cusolver/cublas handle

cusolver_status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);

// step 2: copy A and B to device

cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
cudaStat2 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
cudaStat3 = cudaMalloc ((void**)&devInfo, sizeof(int));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m,
cuMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

// step 3: query working space of syevd

cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvalues and eigenvectors.

cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
cusolver_status = cusolverDnDsyevd_bufferSize{
cusolverH,
jobz,
uplo,
m,
d_A,
lda,
d_W,
&lwork);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

// step 4: compute spectrum

cusolver_status = cusolverDnDsyevd{
cusolverH,
jobz,
uplo,
m,
d_A,
lda,
d_W,
d_work,
lwork,
devInfo};
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);
F.3. Generalized Symmetric-Definite Dense Eigenvalue Solver

This chapter provides a simple example in the C programming language of how to use sygvd to compute spectrum of a pair of dense symmetric matrices \((A, B)\) by

\[
Ax = \lambda Bx
\]

where \(A\) is a 3x3 dense symmetric matrix.
Examples of Dense Eigenvalue Solver

and B is a 3x3 positive definite matrix

\[
A = \begin{pmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2 \\
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
10 & 2 & 3 \\
2 & 10 & 5 \\
3 & 5 & 10 \\
\end{pmatrix}
\]

The following code uses sygvd to compute eigenvalues and eigenvectors, then compare to exact eigenvalues \{0.158660256604, 0.370751508101882, 0.6\}.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++)
    {
        for(int col = 0 ; col < n ; col++)
        {
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    const int m = 3;
    const int lda = m;
    /*
    *       | 3.5 0.5 0 |
    *   A = | 0.5 3.5 0 |
    *       | 0 0 2 |
    *       | 10 2 3 |
    *   B = | 2 10 5 |
    *       | 3 5 10 |
    */
    double A[lda*m] = { 3.5, 0.5, 0, 0.5, 3.5, 0, 0, 0, 2.0};
    double B[lda*m] = { 10.0, 2.0, 3.0, 2.0, 10.0, 5.0, 5.0, 3.0, 5.0, 10.0};
    double lambda[m] = { 0.158660256604, 0.370751508101882, 0.6};
    double *d_A = NULL;
    double *d_B = NULL;
```
double *d_W = NULL;
int *devInfo = NULL;
double *d_work = NULL;
int lwork = 0;
int info_gpu = 0;

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("=====
");
printf("B = (matlab base-1)\n");
printMatrix(m, m, B, lda, "B");
printf("=====
");
call eigenvalue solver

// step 1: create cusolver/cublas handle
    cusolver_status = cusolverDnCreate(&cusolverH);
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);

// step 2: copy A and B to device
    cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
    cudaStat2 = cudaMalloc ((void**)&d_B, sizeof(double) * lda * m);
    cudaStat3 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
    cudaStat4 = cudaMalloc ((void**)&devInfo, sizeof(int));
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
    assert(cudaSuccess == cudaStat4);
    cudaMemcpy(d_A, A, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    cudaMemcpy(d_W, W, sizeof(double) * m, cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);

// step 3: query working space of sygvd
    cusolverEigType_t itype = CUSOLVER_EIG_TYPE_1; // A*x = (lambda)*B*x
    cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
    cusolverStatus = cusolverDnDsygvd_bufferSize(
        cusolverH,
        itype, jobz, uplo, m, d_A, lda, d_B, lda, d_W, &lwork);
    assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);
    cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
    assert(cudaSuccess == cudaStat1);

// step 4: compute spectrum of (A,B)
    cusolverStatus = cusolverDnDsygvd(
        cusolverH,
        itype, jobz, uplo, m,
Examples of Dense Eigenvalue Solver

```
d_A,
lda,
d_B,
lda,
d_W,
d_work,
lwork,
devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);
check the result
cudaStat1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
printf("after sygvd: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);
printf("eigenvalue = (matlab base-1), ascending order\n");
for(int i = 0 ; i < m ; i++){
    printf("W[%d] = %E\n", i+1, W[i]);
}
printf("V = (matlab base-1)\n");
printMatrix(m, m, V, lda, "V");
printf("\n");
// step 4: check eigenvalues
double lambda_sup = 0;
for(int i = 0 ; i < m ; i++){
    double error = fabs( lambda[i] - W[i]);
    lambda_sup = (lambda_sup > error)? lambda_sup : error;
}
printf("|lambda - W| = %E\n", lambda_sup);
// free resources
if (d_A    ) cudaFree(d_A);
if (d_B    ) cudaFree(d_B);
if (d_W    ) cudaFree(d_W);
if (devInfo) cudaFree(devInfo);
if (d_work ) cudaFree(d_work);
    if (cusolverH) cusolverDnDestroy(cusolverH);
    cudaDeviceReset();
    return 0;
}
```
F.4. Generalized Symmetric-Definite Dense Eigenvalue Solver

This chapter provides a simple example in the C programming language of how to use sygvd to compute spectrum of a pair of dense symmetric matrices (A,B) by

$$Ax = \lambda Bx$$

where A is a 3x3 dense symmetric matrix

$$A = \begin{pmatrix} 3.5 & 0.5 & 0 \\ 0.5 & 3.5 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

and B is a 3x3 positive definite matrix

$$B = \begin{pmatrix} 10 & 2 & 3 \\ 2 & 10 & 5 \\ 3 & 5 & 10 \end{pmatrix}$$

The following code uses sygvd to compute eigenvalues and eigenvectors, then compare to exact eigenvalues {0.158660256604, 0.370751508101882, 0.6}.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0; row < m; row++)
    {
        for(int col = 0; col < n; col++)
        {
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    const int m = 3;
    const int lda = m;
    /*
     | 3.5 0.5 0 |
     */
```
Examples of Dense Eigenvalue Solver

```c
* A = | 0.5 3.5 0 |
*     | 0  0  2 |
*     | 10 2 3 |
* B = | 2 10 5 |
*     | 3 5 10 |
*/
double A[lda*m] = { 3.5, 0.5, 0.5, 3.5, 0, 0, 0, 2.0};
double B[lda*m] = { 10.0, 2.0, 3.0, 2.0, 10.0, 5.0, 3.0, 5.0, 10.0};
double lambda[m] = { 0.158660256604, 0.370751508101882, 0.6};

do double V[lda*m]; // eigenvectors
do double W[m]; // eigenvalues
do double *d_A = NULL;
do double *d_B = NULL;
do double *d_W = NULL;
do int *devInfo = NULL;
do double *d_work = NULL;
do int lwork = 0;
do int info_gpu = 0;

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("=====
");
printf("B = (matlab base-1)\n");
printMatrix(m, m, B, lda, "B");
printf("=====
");
call eigenvalue solver

// step 1: create cusolver/cublas handle
 cusolver_status = cusolverDnCreate(&cusolverH);
 assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);

// step 2: copy A and B to device
 cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_W, W, sizeof(double) * m, cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(devInfo, sizeof(int), cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * lda * m, cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(devInfo, sizeof(int), cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

// step 3: query working space of sygvd
 cusolverEigType_t itype = CUSOLVER_EIG_TYPE_1; // A*x = (lambda)*B*x
cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvalues and
 eigenvectors.
cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
cusolver_status = cusolverDnDsygvd_bufferSize(
  cusolverH,
  itype,
  jobz,
  uplo,
  m,
  d_A,
```
lda,  
d_B,  
lda,  
d_W,  
&lwork);  
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);  
cudaStat1 = cudaMemcpy((void**)&d_work, sizeof(double)*lwork);  
assert(cudaSuccess == cudaStat1);  

// step 4: compute spectrum of (A,B)  
cusolver_status = cusolverDnDsygvd(  
cusolverH,  
itype,  
jobz,  
uplo,  
m,  
d_A,  
lda,  
d_B,  
lda,  
d_W,  
d_work,  
lwork,  
devInfo);  
cudaStat1 = cudaMemcpy((void**)&d_work, sizeof(double)*lwork);  
assert(cudaSuccess == cudaStat1);  

check the result

cudaStat1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);  
cudaStat2 = cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);  
cudaStat3 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);  
assert(cudaSuccess == cudaStat1);  
assert(cudaSuccess == cudaStat2);  
assert(cudaSuccess == cudaStat3);  

printf("after sygvd: info_gpu = %d\n", info_gpu);  
assert(0 == info_gpu);  
printf("eigenvalue = (matlab base-1), ascending order\n");  
for(int i = 0 ; i < m ; i++){  
    printf("W[%d] = %E\n", i+1, W[i]);  
}

printf("V = (matlab base-1)\n");  
printMatrix(m, m, V, lda, "V");  
printf("=====");  

// step 4: check eigenvalues  
double lambda_sup = 0;  
for(int i = 0 ; i < m ; i++){  
    double error = fabs( lambda[i] - W[i]);  
    lambda_sup = (lambda_sup > error)? lambda_sup : error;  
}

// free resources  
if (d_A    ) cudaFree(d_A);  
if (d_B    ) cudaFree(d_B);  
if (d_W    ) cudaFree(d_W);  
if (devInfo) cudaFree(devInfo);  
if (d_work ) cudaFree(d_work);  
if (cusolverH) cusolverDnDestroy(cusolverH);
F.5. Standard Symmetric Dense Eigenvalue Solver (via Jacobi method)

This chapter provides a simple example in the C programming language of how to use syevj to compute the spectrum of a dense symmetric system by

\[ Ax = \lambda x \]

where \( A \) is a 3x3 dense symmetric matrix

\[
A = \begin{pmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2
\end{pmatrix}
\]

The following code uses syevj to compute eigenvalues and eigenvectors, then compare to exact eigenvalues \{2,3,4\}.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cudaStream_t stream = NULL;
    syevjInfo_t syevj_params = NULL;

    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    const int m = 3;
    const int lda = m;
    
    cudaDeviceReset();
    return 0;
}
```
Examples of Dense Eigenvalue Solver

```c
/*
A = \[
  3.5  0.5  0 \\
  0.5  3.5  0 \\
  0    0   2
\]
*/

double A[lda*m] = { 3.5, 0.5, 0, 0.5, 3.5, 0, 0, 0, 2.0};
double lambda[m] = { 2.0, 3.0, 4.0};

double V[lda*m]; /* eigenvectors */
double W[m]; /* eigenvalues */
double *d_A = NULL; /* device copy of A */
double *d_W = NULL; /* eigenvalues */
int *d_info = NULL; /* error info */
int lwork = 0; /* size of workspace */
double *d_work = NULL; /* device workspace for syevj */
int info = 0; /* host copy of error info */

/* configuration of syevj */
const double tol = 1.e-7;
const int max_sweeps = 15;
const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvectors.
const cublasFillMode_t  uplo = CUBLAS_FILL_MODE_LOWER;

configure parameters of syevj

/* numerical results of syevj */
double residual = 0;
int executed_sweeps = 0;

printf("example of syevj \
");
printf("tol = %E, default value is machine zero \
", tol);
printf("max. sweeps = %d, default value is 100\n", max_sweeps);

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("=====\n");

/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);

CUDAStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);

status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: configuration of syevj */
status = cusolverDnCreateSyevjInfo(&syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* default value of tolerance is machine zero */
status = cusolverDnXsyevjSetTolerance(
    syevj_params,
    tol);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* default value of max. sweeps is 100 */
status = cusolverDnXsyevjSetMaxSweeps(
    syevj_params,
    max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: copy A to device */
```
cudasmall1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
cudasmall2 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
cudasmall3 = cudaMalloc ((void**)&d_info, sizeof(int));
assert(cudaSuccess == cudasmall1);
assert(cudaSuccess == cudasmall2);
assert(cudaSuccess == cudasmall3);

cudasmall1 = cudaMemcpy(d_A, A, sizeof(double)*lda*m, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudasmall1);

/* step 4: query working space of syevj */
status = cusolverDnDsyej_bufferSize(
    cusolverH,
    jobz,
    uplo,
    m,
    d_A,
    lda,
    d_W,
    &lwork,
    syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudasmall1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudasmall1);

/* step 5: compute eigen-pair */
status = cusolverDnDsyej(
    cusolverH,
    jobz,
    uplo,
    m,
    d_A,
    lda,
    d_W,
    d_work,
    lwork,
    d_info,
    syevj_params);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudasmall1);

cudasmall1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudasmall2 = cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudasmall3 = cudaMemcpy(d_info, &info, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudasmall1);
assert(cudaSuccess == cudasmall2);
assert(cudaSuccess == cudasmall3);

if ( 0 == info ){     
    printf("syej converges \n");
}else if ( 0 > info ){  
    printf("%d-th parameter is wrong \n", -info);
    exit(1);
}else{     
    printf("WARNING: info = %d : syevj does not converge \n", info );
}

printf("Eigenvalue = (matlab base-1), ascending order\n");
for(int i = 0 ; i < m ; i++){ 
    printf("W[\%d] = %E\n", i+1, W[i]);
}

This chapter provides a simple example in the C programming language of how to use `sygvj` to compute spectrum of a pair of dense symmetric matrices \( (A,B) \) by

\[ Ax = \lambda Bx \]

where \( A \) is a 3x3 dense symmetric matrix
Examples of Dense Eigenvalue Solver

\[
A = \begin{pmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2
\end{pmatrix}
\]

and \( B \) is a 3x3 positive definite matrix

\[
B = \begin{pmatrix}
10 & 2 & 3 \\
2 & 10 & 3 \\
3 & 5 & 10
\end{pmatrix}
\]

The following code uses \texttt{sygvj} to compute eigenvalues and eigenvectors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
  for(int row = 0 ; row < m ; row++){
    for(int col = 0 ; col < n ; col++){
      double Areg = A[row + col*lda];
      printf("%s(%d,%d) = %f
", name, row+1, col+1, Areg);
    }
  }
}

int main(int argc, char*argv[])
{
  cusolverDnHandle_t cusolverH = NULL;
  cudaStream_t stream = NULL;
  syevjInfo_t syevj_params = NULL;
  cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
  cudaError_t cudaStat1 = cudaSuccess;
  cudaError_t cudaStat2 = cudaSuccess;
  cudaError_t cudaStat3 = cudaSuccess;
  cudaError_t cudaStat4 = cudaSuccess;
  const int m = 3;
  const int lda = m;
  double A[lda*m] = { 3.5, 0.5, 0, 0.5, 3.5, 0, 0, 0, 2.0};
  double B[lda*m] = { 10.0, 2.0, 3.0, 2.0, 10.0, 5.0, 3.0, 5.0, 10.0};
  double lambda[m] = { 0.158660256604, 0.370751508101882, 0.6};
  double *d_A = NULL; /* device copy of A */
  double *d_B = NULL; /* device copy of B */
```
```
### Examples of Dense Eigenvalue Solver

**cuSOLVER Library**

```c
double *d_W = NULL; /* numerical eigenvalue */
int *d_info = NULL; /* error info */
int lwork = 0; /* size of workspace */
double *d_work = NULL; /* device workspace for sygvj */
int info = 0; /* host copy of error info */
```

**configure parameters of Jacobi method**

```c
/* configuration of sygvj */
const double tol = 1.e-7;
const int max_sweeps = 15;
const cusolverEigType_t itype = CUSOLVER_EIG_TYPE_1; // A*x = (lambda)*B*x
cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvectors.
const cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;

/* numerical results of syevj */
double residual = 0;
int executed_sweeps = 0;
```

```c
/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: configuration of syevj */
status = cusolverDnCreateSyevjInfo(&syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);
status = cusolverDnXsyevjSetTolerance(syevj_params, tol);
assert(CUSOLVER_STATUS_SUCCESS == status);
status = cusolverDnXsyevjSetMaxSweeps(syevj_params, max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: copy A and B to device */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
cudaStat2 = cudaMalloc ((void**)&d_B, sizeof(double) * lda * m);
```
Examples of Dense Eigenvalue Solver

cudaStat3 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
cudaStat4 = cudaMalloc ((void**)&d_info, sizeof(int));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m,
cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * lda * m,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

/* step 4: query working space of sygvj */
status = cusolverDnDsygvj_bufferSize(
cusolverH,
itype,
jobz,
uplo,
m,
d_A,
lda,
d_B,
lda, /* ldb */
d_W,
&lwork,
syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

/* step 5: compute spectrum of (A,B) */
status = cusolverDnDsygvj(
cusolverH,
itype,
jobz,
uplo,
m,
d_A,
lda,
d_B,
lda, /* ldb */
d_W,
d_work,
lwork,
d_info,
syevj_params);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(&info, d_info, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

check the result

if ( 0 == info ){
  printf("sygvj converges \n");
} else if ( 0 > info ){
  printf("Error: %d-th parameter is wrong \n", -info);
}
```c
exit(1);
}
else if ( m >= info ) {
    printf("Error: leading minor of order %d of B is not positive definite\n", -info);
    exit(1);
} else { /* info = m+1 */
    printf("WARNING: info = %d : sygvj does not converge \n", info );
}

printf("Eigenvalue = (matlab base-1), ascending order\n");
for(int i = 0 ; i < m ; i++){
    printf("W[%d] = %E\n", i+1, W[i]);
}

printf("V = (matlab base-1)\n");
printMatrix(m, m, V, lda, "V");
printf("====\n");
/* step 6: check eigenvalues */
    double lambda_sup = 0;
    for(int i = 0 ; i < m ; i++){
        double error = fabs( lambda[i] - W[i]);
        lambda_sup = (lambda_sup > error)? lambda_sup : error;
    }
    printf("|lambda - W| = %E\n", lambda_sup);

    status = cusolverDnXsyevjGetSweeps(
        cusolverH,
        syevj_params,
        &executed_sweeps);
    assert(CUSOLVER_STATUS_SUCCESS == status);

    status = cusolverDnXsyevjGetResidual(
        cusolverH,
        syevj_params,
        &residual);
    assert(CUSOLVER_STATUS_SUCCESS == status);

    printf("residual |M - V*W*V**H|_F = %E \n", residual );
    printf("number of executed sweeps = %d \n", executed_sweeps );

/* free resources */
if (d_A    ) cudaFree(d_A);
if (d_B    ) cudaFree(d_B);
if (d_W    ) cudaFree(d_W);
if (d_info ) cudaFree(d_info);
if (d_work ) cudaFree(d_work);
if (cusolverH) cusolverDnDestroy(cusolverH);
if (stream   ) cudaStreamDestroy(stream);
if (syevj_params) cusolverDnDestroySyevjInfo(syevj_params);
    cudaDeviceReset();
    return 0;
}
```

F.7. batch eigenvalue solver for dense symmetric matrix

This chapter provides a simple example in the C programming language of how to use syevjBatched to compute the spectrum of a sequence of dense symmetric matrices by
$A_j x = \lambda x$

where $A_0$ and $A_1$ are 3x3 dense symmetric matrices

$$A_0 = \begin{bmatrix}
1 & -1 & 0 \\
-1 & 2 & 0 \\
0 & 0 & 0
\end{bmatrix},

A_1 = \begin{bmatrix}
3 & 4 & 0 \\
4 & 7 & 0 \\
0 & 0 & 0
\end{bmatrix}$$

The following code uses syevjBatched to compute eigenvalues and eigenvectors

$$A_j = V_j W_j V_j^T$$

The user can disable/enable sorting by the function cusolverDnXsyevjSetSortEig.
Examples of Dense Eigenvalue Solver

```
* W0 = diag(0, 0.3820, 2.6180)
* A1 = | 3 4 0 |
*     | 4 7 0 |
*     | 0 0 0 |
* A1 = V1 * W1 * V1**T
* W1 = diag(0, 0.5279, 9.4721)
*/

setup matrices A0 and A1

double A[lda*m*batchSize]; /* A = [A0 ; A1] */
double V[lda*m*batchSize]; /* V = [V0 ; V1] */
double W[m*batchSize];     /* W = [W0 ; W1] */
int info[batchSize];       /* info = [info0 ; info1] */
double *d_A  = NULL; /* lda-by-m-by-batchSize */
double *d_W  = NULL; /* m-by-batchSize */
int *d_info  = NULL; /* batchSize */
lwork = 0;  /* size of workspace */
double *d_work = NULL; /* device workspace for syevjBatched */

const double tol = 1.e-7;
const int max_sweeps = 15;
const int sort_eig  = 0;   /* don't sort eigenvalues */
const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; /* compute eigenvectors */
const cublasFillMode_t  uplo = CUBLAS_FILL_MODE_LOWER;

/* residual and executed_sweeps are not supported on syevjBatched */
double residual = 0;
int executed_sweeps = 0;

double *A0 = A;
double *A1 = A + lda*m;
/*
* A0 = | 1 -1 0 |
*     | -1 2 0 |
*     | 0 0 0 |
* A0 is column-major */
A0[0 + 0*lda] = 1.0;
A0[1 + 0*lda] = -1.0;
A0[2 + 0*lda] = 0.0;
A0[0 + 1*lda] = -1.0;
A0[1 + 1*lda] = 2.0;
A0[2 + 1*lda] = 0.0;
A0[0 + 2*lda] = 0.0;
A0[1 + 2*lda] = 0.0;
A0[2 + 2*lda] = 0.0;
/*
* A1 = | 3 4 0 |
*     | 4 7 0 |
*     | 0 0 0 |
* A1 is column-major */
A1[0 + 0*lda] = 3.0;
A1[1 + 0*lda] = 4.0;
A1[2 + 0*lda] = 0.0;
A1[0 + 1*lda] = 4.0;
```
configure parameters of syevj

```c
/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);

status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: configuration of syevj */
status = cusolverDnCreateSyevjInfo(&syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* default value of tolerance is machine zero */
status = cusolverDnXsyevjSetTolerance(syevj_params, tol);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* default value of max. sweeps is 100 */
status = cusolverDnXsyevjSetMaxSweeps(syevj_params, max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* disable sorting */
status = cusolverDnXsyevjSetSortEig(syevj_params, sort_eig);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: copy A to device */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m * batchSize);
cudaStat2 = cudaMalloc ((void**)&d_W, sizeof(double) * m * batchSize);
cudaStat3 = cudaMalloc ((void**)&d_info, sizeof(int   ) * batchSize);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
cudaStat1 = cudaMemcpy(d_A, A, sizeof(double) * lda * m * batchSize, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_W, W, sizeof(double) * m * batchSize, cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_info, info, sizeof(int   ) * batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

/* step 4: query working space of syevjBatched */
status = cusolverDnDsyevjbBatched_bufferSize(cusolverH, jobz, uplo, m, d_A, lda, d_W, &lwork, syevj_params,
```

Examples of Dense Eigenvalue Solver

```c
A1[1 + 1*lda] = 7.0;
A1[2 + 1*lda] = 0.0;
A1[0 + 2*lda] = 0.0;
A1[1 + 2*lda] = 0.0;
A1[2 + 2*lda] = 0.0;
```
call eigenvalue solver

/* step 5: compute spectrum of A0 and A1 */
    status = cusolverDnDsyevjbapped(cusolverH, jobz, uplo,
        m, d_A, lda, d_W, d_work, lwork, d_info,
        syevj_params, batchSize);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSOLVER_STATUS_SUCCESS == status);
    assert(cudaSuccess == cudaStat1);
    cudaStat2 = cudaMemcpy(W , d_A , sizeof(double) * lda * m * batchSize, cudaMemcpyDeviceToHost);
    cudaStat3 = cudaMemcpy(W , d_W , sizeof(double) * m * batchSize , cudaMemcpyDeviceToHost);
    cudaStat4 = cudaMemcpy(&info, d_info, sizeof(int) * batchSize , cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
    for(int i = 0 ; i < batchSize ; i++){
        if ( 0 == info[i] ){
            printf("matrix %d: syevj converges \
", i);
        }else if ( 0 > info[i] ){/* only info[0] shows if some input parameter is wrong. */
            printf("Error: %d-th parameter is wrong \
", -info[i] );
            exit(1);
        }else { /* info = m+1 */
            printf("WARNING: matrix %d, info = %d : sygvj does not converge \n", i, info[i] );
        }
    }

/* Step 6: show eigenvalues and eigenvectors */
    double *W0 = W;
    double *W1 = W + m;
    printf("\n");
    for(int i = 0 ; i < m ; i++){
        printf("W0[%d] = %f\n", i, W0[i]);
    }
    printf("\n");
    for(int i = 0 ; i < m ; i++){
        printf("W1[%d] = %f\n", i, W1[i]);
    }
    printf("\n");
cannot query residual and executed sweeps.

```
double *V0 = V;
double *V1 = V + lda*m;
printf("V0 = (matlab base-1)\n");
printMatrix(m, m, V0, lda, "V0");
printf("V1 = (matlab base-1)\n");
printMatrix(m, m, V1, lda, "V1");
```

```c
/* The following two functions do not support batched version. 
* The error CUSOLVER_STATUS_NOT_SUPPORTED is returned. */
status = cusolverDnXsyevjGetSweeps(
    cusolverH,
    syevj_params,
    &executed_sweeps);
assert(CUSOLVER_STATUS_NOT_SUPPORTED == status);

status = cusolverDnXsyevjGetResidual(
    cusolverH,
    syevj_params,
    &residual);
assert(CUSOLVER_STATUS_NOT_SUPPORTED == status);

/* free resources */
if (d_A    ) cudaFree(d_A);
if (d_W    ) cudaFree(d_W);
if (d_info ) cudaFree(d_info);
if (d_work ) cudaFree(d_work);

if (cusolverH) cusolverDnDestroy(cusolverH);
if (stream    ) cudaStreamDestroy(stream);
if (syevj_params) cusolverDnDestroySyevjInfo(syevj_params);

cudaDeviceReset();

return 0;

```
Appendix G. Examples of Singular Value Decomposition

G.1. SVD with singular vectors

This chapter provides a simple example in the C programming language of how to perform singular value decomposition.

\[ A = U \Sigma V^H \]

A is a 3x2 dense matrix,

\[
A = \begin{pmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0 \\
\end{pmatrix}
\]

The following code uses three steps:
Step 1: compute \( A = U\Sigma V^T \)
Step 2: check accuracy of singular value
Step 3: measure residual \( A-U\Sigma V^T \)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name) {
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            printf("%f ", A[row*lda+col]);
        }
        printf("\n");
    }
}
```

...
```c
int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cublasHandle_t cublasH = NULL;
    cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    cudaError_t cudaStat6 = cudaSuccess;
    const int m = 3;
    const int n = 2;
    const int lda = m;

    const double A[lda*n] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0};
    double U[lda*m]; // m-by-m unitary matrix
    double VT[lda*n];  // n-by-n unitary matrix
    double S[n]; // singular value
    double S_exact[n] = {7.065283497082729, 1.040081297712078};
    double *d_A = NULL;
    double *d_S = NULL;
    double *d_U = NULL;
    double *d_VT = NULL;
    int *devInfo = NULL;
    double *d_W = NULL;  // W = S*VT
    int lwork = 0;
    int info_gpu = 0;
    const double h_one = 1;
    const double h_minus_one = -1;

    printf("A = (matlab base-1)\n");
    printMatrix(m, n, A, lda, "A");
    printf("=====
");

    // step 1: create cusolverDn/cublas handle
    cusolver_status = cusolverDnCreate(&cusolverH);
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
    cublas_status = cublasCreate(&cublasH);
    assert(CUBLAS_STATUS_SUCCESS == cublas_status);

    // step 2: copy A and B to device
    cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double)*lda*n);
    cudaStat2 = cudaMalloc ((void**)&d_S, sizeof(double)*n);
    cudaStat3 = cudaMalloc ((void**)&d_U, sizeof(double)*lda*m);
    cudaStat4 = cudaMalloc ((void**)&d_VT, sizeof(double)*lda*n);
    cudaStat5 = cudaMalloc ((void**)&devInfo, sizeof(int));
    cudaStat6 = cudaMalloc ((void**)&d_W, sizeof(double)*lda*n);
    ...
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);
assert(cudaSuccess == cudaStat6);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

// step 3: query working space of SVD
cusolver_status = cusolverDnDgesvd_bufferSize(
    cusolverH,
    m, n,
    &lwork);
assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

cudaStat1 = cudaMalloc((void**)&d_work , sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

// step 4: compute SVD
signed char jobu = 'A'; // all m columns of U
signed char jobvt = 'A'; // all n columns of VT
cusolver_status = cusolverDnDgesvd (
    cusolverH,
    jobu, jobvt, m, n, d_A, lda, d_S, lda, d_U, lda, // ldu
    d_VT, lda, // ldvt,
    d_work, lwork,
    d_rwork, d_rwork,
    devInfo);
cudaStat1 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(VT, d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S , d_S , sizeof(double)*n    , cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);

printf("after gesvd: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);
printf("=====
");
printf("S = (matlab base-1)\n");
printMatrix(n, 1, S, lda, "S");
printf("====="
);
print("U = (matlab base-1)\n");
printMatrix(m, m, U, lda, "U");
printf("====\n");

print("VT = (matlab base-1)\n");
printMatrix(n, n, VT, lda, "VT");
printf("====\n");

// step 5: measure error of singular value
double ds_sup = 0;
for(int j = 0; j < n; j++){
    double err = fabs(S[j] - S_exact[j]);
    ds_sup = (ds_sup > err)? ds_sup : err;
}
printf("|S - S_exact| = %E\n", ds_sup);

// step 6: |A - U*S*VT|
// W = S*VT
   cublas_status = cublasDdgmm(
      cublasH,
      CUBLAS_SIDE_LEFT,
      n,
      n,
      d_VT,
      lda,
      d_S,
      1,
      d_W,
      lda);
   assert(CUBLAS_STATUS_SUCCESS == cublas_status);

   // A := -U*W + A
   cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
   assert(cudaSuccess == cudaMemcpy);
   cublas_status = cublasDgemm_v2(
      cublasH,
      CUBLAS_OP_N, // U
      CUBLAS_OP_N, // W
      m, // number of rows of A
      n, // number of columns of A
      n, // number of columns of U
      &h_minus_one, /* host pointer */
      d_U, // U
      lda,
      d_W, /* hostpointer */
      lda,
      &d_R_fro,
      d_A,
      lda);
   assert(CUBLAS_STATUS_SUCCESS == cublas_status);
   printf("|A - U*S*VT| = %E\n", dR_fro);

   // free resources
   if (d_A    ) cudaFree(d_A);
   if (d_S    ) cudaFree(d_S);
   if (d_U    ) cudaFree(d_U);
   if (d_VT   ) cudaFree(d_VT);
   if (devInfo) cudaFree(devInfo);
Examples of Singular Value Decomposition

G.2. **SVD with singular vectors (via Jacobi method)**

This chapter provides a simple example in the C programming language of how to perform singular value decomposition by gesvdj.

\[ A = U \Sigma V^H \]

A is a 3x2 dense matrix,

\[
\begin{pmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0
\end{pmatrix}
\]

...
Examples of Singular Value Decomposition

cudaError_t cudaStat1 = cudaSuccess;
cudaError_t cudaStat2 = cudaSuccess;
cudaError_t cudaStat3 = cudaSuccess;
cudaError_t cudaStat4 = cudaSuccess;
cudaError_t cudaStat5 = cudaSuccess;
const int m = 3;
const int n = 2;
const int lda = m;
/*       | 1 2 |
   A = | 4 5 |
       | 2 1 |
*/
double A[lda*n] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0};
double U[lda*m]; /* m-by-m unitary matrix, left singular vectors */
double V[lda*n]; /* n-by-n unitary matrix, right singular vectors */
double S[n];     /* numerical singular value */
/* exact singular values */
double S_exact[n] = {7.065283497082729, 1.040081297712078};
double *d_A = NULL;  /* device copy of A */
double *d_S = NULL;  /* singular values */
double *d_U = NULL; /* left singular vectors */
double *d_V = NULL; /* right singular vectors */
int *d_info = NULL;  /* error info */
int lwork = 0;       /* size of workspace */
double *d_work = NULL; /* device workspace for gesvdj */
int info = 0;        /* host copy of error info */

/* configuration of gesvdj */
const double tol = 1.e-7;
const int max_sweeps = 15;
const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute

/* numerical results of gesvdj */
double residual = 0;
int executed_sweeps = 0;
printf("example of gesvdj \n");
printf("tol = %E, default value is machine zero \n", tol);
printf("max. sweeps = %d, default value is 100\n", max_sweeps);
printf("econ = %d \n", econ);
printf("A = (matlab base-1)\n");
printMatrix(m, n, A, lda, "A");
printf("==\n");

/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 2: configuration of gesvdj */
status = cusolverDnCreateGesvdjInfo(&gesvdj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* default value of tolerance is machine zero */
status = cusolverDnXgesvdjSetTolerance(
Examples of Singular Value Decomposition

gesvdj_params,
tol);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* default value of max. sweeps is 100 */
status = cusolverDnXgesvdjSetMaxSweeps(
    gesvdj_params,
    max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: copy A and B to device */
cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
...
Examples of Singular Value Decomposition

d_U,  /* m-by-m if econ = 0 */
/* m-by-min(m,n) if econ = 1 */
lda,   /* leading dimension of U, ldu >= max(1,m) */
d_V,   /* n-by-n if econ = 0 */
/* n-by-min(m,n) if econ = 1 */
llda,  /* leading dimension of V, ldv >= max(1,n) */
d_work,
l_work,
d_info,
gesvdj_params);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);
cudaStat2 = cudaMemcpy(U, d_U, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(V, d_V, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info, d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat5 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);

if ( 0 == info ){
  printf("gesvdj converges \n");
} else if ( 0 > info ){
  printf("%d-th parameter is wrong \n", -info);
  exit(1);
} else{
  printf("WARNING: info = %d : gesvdj does not converge \n", info );
}

printf("S = singular values (matlab base-1)\n");
printMatrix(n, 1, S, lda, "S");
printf("====\n");

printf("U = left singular vectors (matlab base-1)\n");
printMatrix(m, m, U, lda, "U");
printf("====\n");

printf("V = right singular vectors (matlab base-1)\n");
printMatrix(n, n, V, lda, "V");
printf("====\n");

/* step 6: measure error of singular value */
double ds_sup = 0;
for(int j = 0; j < n; j++){
  double err = fabs(S[j] - S_exact[j]);
  ds_sup = (ds_sup > err)? ds_sup : err;
}  
printf("|S - S_exact|_sup = %E \n", ds_sup);

status = cusolverDnXgesvdjGetSweeps(
cusolverH,
gesvdj_params,
&executed_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

status = cusolverDnXgesvdjGetResidual(
cusolverH,
gesvdj_params,
&residual);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("residual \|A - U*S*V**H\|_F = %E \n", residual);
printf("number of executed sweeps = %d \n", executed_sweeps);

/* free resources */
if (d_A) cudaFree(d_A);
if (d_S) cudaFree(d_S);
if (d_U) cudaFree(d_U);
if (d_V) cudaFree(d_V);
if (d_info) cudaFree(d_info);
if (d_work) cudaFree(d_work);

if (cusolverH) cusolverDnDestroy(cusolverH);
if (stream) cudaStreamDestroy(stream);
if (gesvdj_params) cusolverDnDestroyGesvdjInfo(gesvdj_params);

cudaDeviceReset();
return 0;
}

G.3. **batch dense SVD solver**

This chapter provides a simple example in the C programming language of how to use gesvdjBatched to compute the SVD of a sequence of dense matrices

\[ A_j = U_j \Sigma_j V_j^H \]

where \( A_0 \) and \( A_1 \) are 3x2 dense matrices

\[
A_0 = \begin{pmatrix}
1 & -1 \\
-1 & 2 \\
0 & 0
\end{pmatrix}
\]

\[
A_1 = \begin{pmatrix}
3 & 4 \\
4 & 7 \\
0 & 0
\end{pmatrix}
\]

The following code uses gesvdjBatched to compute singular values and singular vectors.

The user can disable/enable sorting by the function cusolverDnXgesvdjSetSortEig.

/*
* How to compile (assume cuda is installed at /usr/local/cuda/)
* nvcc -c -I/usr/local/cuda/include gesvdjbatch_example.cpp
* g++ -o gesvdjbatch_example gesvdjbatch_example.o -L/usr/local/cuda/lib64 -lcusolver -lcudart
*/
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const double* A, int lda, const char* name) {
  for (int row = 0 ; row < m ; row++) {
    for (int col = 0 ; col < n ; col++) {
      printf("%f", A[row * lda + col]);
      if (col < n - 1) printf(" ");
    }
    printf("\n");
  }
  printf("\n");
}
double Areg = A[row + col*lda];
printf("%s(%d,%d) = %20.16E\n", name, row+1, col+1, Areg);
}
}
}
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
cudaStream_t stream = NULL;
gesvdjInfo_t gesvdj_params = NULL;

cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;
cudaError_t cudaStat2 = cudaSuccess;
cudaError_t cudaStat3 = cudaSuccess;
cudaError_t cudaStat4 = cudaSuccess;
cudaError_t cudaStat5 = cudaSuccess;
    const int m = 3; /* 1 <= m <= 32 */
    const int n = 2; /* 1 <= n <= 32 */
    const int lda = m; /* lda >= m */
    const int ldu = m; /* ldu >= m */
    const int ldv = n; /* ldv >= n */
    const int batchSize = 2;
    const int minmn = (m < n)? m : n; /* min(m,n) */

    int info[batchSize];       /* info = [info0 ; info1] */
    double *d_A  = NULL; /* lda-by-n-by-batchSize */
    double *d_U  = NULL; /* ldu-by-m-by-batchSize */
    double *d_V  = NULL; /* ldv-by-n-by-batchSize */
    double *d_S  = NULL; /* minmn-by-batchSize */
    int*  d_info  = NULL; /* batchSize */
    int lwork = 0;       /* size of workspace */
    double *d_work = NULL; /* device workspace for gesvdjBatched */
    const double tol1 = 1.e-7;
    const int max_sweeps = 15;
    const int sort_svd  = 0;   /* don't sort singular values */
    const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; /* compute singular vectors */

    /* residual and executed_sweeps are not supported on gesvdjBatched */
    double residual = 0;
    int executed_sweeps = 0;

setup matrices A0 and A1

    double A[lda*n*batchSize]; /* A = [A0 ; A1] */
    double U[ldu*m*batchSize]; /* U = [U0 ; U1] */
    double V[ldv*n*batchSize]; /* V = [V0 ; V1] */
    double S[minmn*batchSize]; /* S = [S0 ; S1] */
    double A0[lda*n*batchSize]; /* A = [A0 ; A1] */
    double U0[ldu*m*batchSize]; /* U = [U0 ; U1] */
    double V0[ldv*n*batchSize]; /* V = [V0 ; V1] */
    double S0[minmn*batchSize]; /* S = [S0 ; S1] */
    double A1[lda*n*batchSize]; /* A = [A0 ; A1] */
    double U1[ldu*m*batchSize]; /* U = [U0 ; U1] */
    double V1[ldv*n*batchSize]; /* V = [V0 ; V1] */
    double S1[minmn*batchSize]; /* S = [S0 ; S1] */
    double S0 = diag(2.6180, 0.382)
    double S1 = diag(9.4721, 0.5279)
Examples of Singular Value Decomposition

double *A0 = A;
double *A1 = A + lda*n; /* Aj is m-by-n */
/*
 *        |  1  -1  |
 *   A0 = | -1   2  |
 *        |  0   0  |
 *   A0 is column-major
 */
A0[0 + 0*lda] =  1.0;
A0[1 + 0*lda] = -1.0;
A0[2 + 0*lda] =  0.0;
A0[0 + 1*lda] = -1.0;
A0[1 + 1*lda] =  2.0;
A0[2 + 1*lda] =  0.0;
/*
 *        |  3   4  |
 *   A1 = |  4   7  |
 *        |  0   0  |
 *   A1 is column-major
 */
A1[0 + 0*lda] =  3.0;
A1[1 + 0*lda] =  4.0;
A1[2 + 0*lda] =  0.0;
A1[0 + 1*lda] =  4.0;
A1[1 + 1*lda] =  7.0;
A1[2 + 1*lda] =  0.0;
printf("example of gesvdjBatched \n");
printf("m = %d, n = %d \n", m, n);
printf("tol = %E, default value is machine zero \n", tol);
printf("max. sweeps = %d, default value is 100\n", max_sweeps);
configure parameters of gesvdj

/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* step 2: configuration of gesvdj */
status = cusolverDnCreateGesvdjInfo(&gesvdj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* default value of tolerance is machine zero */
status = cusolverDnXgesvdjSetTolerance(
gesvdj_params,
tol);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* default value of max. sweeps is 100 */
status = cusolverDnXgesvdjSetMaxSweeps(
Examples of Singular Value Decomposition

gesvdj_params,
max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* disable sorting */
status = cusolverDnXgesvdjSetSortEig(
gesvdj_params,
sort_svd);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: copy A to device */
cudaStat1 = cudaMalloc ((void**)&d_A   , sizeof(double)*lda*n*batchSize);
cudaStat2 = cudaMalloc ((void**)&d_U   , sizeof(double)*ldu*m*batchSize);
cudaStat3 = cudaMalloc ((void**)&d_V   , sizeof(double)*ldv*n*batchSize);
cudaStat4 = cudaMalloc ((void**)&d_S   , sizeof(double)*minmn*batchSize);
cudaStat5 = cudaMalloc ((void**)&d_info, sizeof(int   )*batchSize);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n*batchSize,
cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_U, U, sizeof(double)*ldu*m*batchSize,
cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_V, V, sizeof(double)*ldv*n*batchSize,
cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(d_S, S, sizeof(double)*minmn*batchSize,
cudaMemcpyHostToDevice);
cudaStat5 = cudaMemcpy(d_info, info, sizeof(int   )*batchSize,
cudaMemcpyHostToDevice);
cudaStat6 = cudaMemcpy(d_U, U, sizeof(double)*ldu*m*batchSize,
cudaMemcpyHostToDevice);
cudaStat7 = cudaMemcpy(d_V, V, sizeof(double)*ldv*n*batchSize,
cudaMemcpyHostToDevice);
cudaStat8 = cudaMemcpy(d_S, S, sizeof(double)*minmn*batchSize,
cudaMemcpyHostToDevice);

/* step 4: query working space of gesvdjBatched */
status = cusolverDnDgesvdjBatched_bufferSize(
cusolverH,
jobz,
m,
n,
d_A,
lda,
d_S,
d_U,
ldu,
d_V,
ldv,
&lwork,
gesvdj_params,
batchSize);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

/* step 5: compute singular values of A0 and A1 */
status = cusolverDnDgesvdjBatched(
cusolverH,
jobz,
m,
n,
d_A,
lda,
d_S,
d_U,
ldu,
d_V,
ldv,
Examples of Singular Value Decomposition

cusolver

check the result

```c
for(int i = 0 ; i < batchSize ; i++){
    if ( 0 == info[i] ){
        printf("matrix %d: gesvdj converges \n", i);
    }else if ( 0 > info[i] ){ /* only info[0] shows if some input parameter is wrong. */
        if so, the error is CUSOLVER_STATUS_INVALID_VALUE.
        printf("Error: %d-th parameter is wrong \n", -info[i] );
        exit(1);
    }else { /* info = m+1 */
        printf("WARNING: matrix %d, info = %d : gesvdj does not converge \n", i, info[i] );
    }
}
```

/* Step 6: show singular values and singular vectors */

double *S0 = S;
double *S1 = S + minmn;
printf("==== \n");
for(int i = 0 ; i < minmn ; i++){
    printf("S0(%d) = %20.16E\n", i+1, S0[i]);
}
printf("==== \n");
for(int i = 0 ; i < minmn ; i++){
    printf("S1(%d) = %20.16E\n", i+1, S1[i]);
}
printf("==== \n");
double *U0 = U;
double *U1 = U + ldu*m; /* Uj is m-by-m */
printf("U0 = (matlab base-1)\n");
printMatrix(m, m, U0, ldu, "U0");
printf("U1 = (matlab base-1)\n");
printMatrix(m, m, U1, ldu, "U1");

double *V0 = V;
double *V1 = V + ldv*n; /* Vj is n-by-n */
printf("V0 = (matlab base-1)\n");
```

```
```
G.4. SVD approximation

This chapter provides a simple example in the C programming language of how to approximate singular value decomposition by `gesvdaStridedBatched`. 

\[ A = U \Sigma V^H \]

A0 and A1 are a 3x2 dense matrices,

\[ A0 = \begin{pmatrix} 1.0 & 2.0 \\ 4.0 & 5.0 \\ 2.0 & 1.0 \end{pmatrix} \]

\[ A1 = \begin{pmatrix} 10.0 & 9.0 \\ 8.0 & 7.0 \\ 6.0 & 5.0 \end{pmatrix} \]
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusolverDn.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0; row < m; row++)
    {
        for(int col = 0; col < n; col++)
        {
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %20.16E\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cudaStream_t stream = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int batchSize = 2;
    const int m = 3;
    const int n = 2;
    const int lda = m;
    const int ldu = m;
    const int ldv = n;
    const int rank = n;
    const long long int strideA = (long long int)lda*n;
    const long long int strideS = n;
    const long long int strideU = (long long int)ldu*n;
    const long long int strideV = (long long int)ldv*n;
/*        | 1 2  |       | 10 9 |
*   A0 = | 4 5 |
*/
    float A[strideA*batchSize] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0, 10.0, 8.0, 6.0, 9.0, 7.0, 5.0};
    float U[strideU*batchSize]; /* left singular vectors */
    float V[ strideV*batchSize]; /* right singular vectors */
    float S[ strideS*batchSize]; /* numerical singular value */
/* exact singular values */
    float S_exact[ strideS*batchSize] = { 7.065283497082729, 1.040081297712078, 18.839649186929730, 0.260035600289472};
```
Examples of Singular Value Decomposition

```c
float *d_S = NULL; /* singular values */
float *d_U = NULL; /* left singular vectors */
float *d_V = NULL; /* right singular vectors */
int *d_info = NULL; /* error info */
int lwork = 0; /* size of workspace */
float *d_work = NULL; /* device workspace for gesvda */
const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute
eigenvectors.
double RnrmF[batchSize]; /* residual norm */
int info[batchSize]; /* host copy of error info */

printf("example of gesvdaStridedBatched \n");
printf("A = (matlab base-1)\n");
printMatrix(m, n, A, lda, "A0");
printf("=====
");
printMatrix(m, n, A + strideA, lda, "A1");
printf("=====
");
/* step 1: create cusolver handle, bind a stream */
status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusolverDnSetStream(cusolverH, stream);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* step 2: copy A to device */
cudaStat1 = cudaMalloc ((void**)&d_A   , sizeof(float)*strideA*batchSize);
cudaStat2 = cudaMalloc ((void**)&d_S   , sizeof(float)*strideS*batchSize);
cudaStat3 = cudaMalloc ((void**)&d_U   , sizeof(float)*strideU*batchSize);
cudaStat4 = cudaMalloc ((void**)&d_V   , sizeof(float)*strideV*batchSize);
cudaStat5 = cudaMalloc ((void**)&d_info, sizeof(int)*batchSize);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(float)*strideA*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize(); /* sync with null stream */
/* step 3: query workspace of SVD */
status = cusolverDnSgesvdaStridedBatched_bufferSize(cusolverH,
jobz, /* CUSOLVER_EIG_MODE_NOVECTOR: compute singular values only */
/* CUSOLVER_EIG_MODE_VECTOR: compute singular value and singular
vectors */
rank, /* number of singular values */
m, /* number of rows of Aj, 0 <= m */
n, /* number of columns of Aj, 0 <= n */
d_A, /* Aj is m-by-n */
lda, /* leading dimension of Aj */
strideA, /* >= lda*n */
d_S, /* Sj is rank-by-1, singular values in descending order */
strideS, /* >= rank */
d_U, /* Uj is m-by-rank */
ldu, /* leading dimension of Uj, ldu >= max(1,m) */
strideU, /* >= ldu*rank */
d_V, /* Vj is n-by-rank */
ldv, /* leading dimension of Vj, ldv >= max(1,n) */
strideV, /* >= ldv*rank */
lwork,
batchSize /* number of matrices */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
```
... 

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(float)*lwork);
assert(cudaSuccess == cudaStat1);

/* step 4: compute SVD */
status = cusolverDnSgesvdaStridedBatched(
cusolverH,
jobz, /* CUSOLVER_EIG_MODE_NOVECTOR: compute singular values only */
/* CUSOLVER_EIG_MODE_VECTOR: compute singular value and singular 
  vectors */
rank, /* number of singular values */
m,    /* number of rows of Aj, 0 <= m */
n,    /* number of columns of Aj, 0 <= n */
d_A,  /* Aj is m-by-n */
lda,  /* leading dimension of Aj */
strideA, /* >= lda*n */
d_S,  /* Sj is rank-by-1 */
/* the singular values in descending order */
strideS, /* => rank */
d_U,  /* Uj is m-by-rank */
ldu,  /* leading dimension of Uj, ldu >= max(1,m) */
strideU, /* => ldu*rank */
d_V,  /* Vj is n-by-rank */
ldv,  /* leading dimension of Vj, ldv >= max(1,n) */
strideV, /* => ldv*rank */
d_work,
lwork,
d_info,
RnrmF,
batchSize /* number of matrices */
);

cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(U, d_U, sizeof(float)*strideU*batchSize,
cudaMemcpyDeviceToHost);

cudaStat2 = cudaMemcpy(V, d_V, sizeof(float)*strideV*batchSize,
cudaMemcpyDeviceToHost);

cudaStat3 = cudaMemcpy(S, d_S, sizeof(float)*strideS*batchSize,
cudaMemcpyDeviceToHost);

cudaStat4 = cudaMemcpy(info, d_info, sizeof(int)*batchSize,
cudaMemcpyDeviceToHost);

cudaStat5 = cudaMemcpy(info, d_info, sizeof(int)*batchSize,
cudaMemcpyDeviceToHost);

if ( 0 > info[0] ){
  printf("%d-th parameter is wrong \n", -info[0]);
  exit(1);
}

for(int idx = 0 ; idx < batchSize; idx++){
  if ( 0 == info[idx] ){
    printf("%d-th matrix, gesvda converges \n", idx );
  }else{
    printf("WARNING: info[%d] = %d : gesvda does not converge \n", idx, info[idx] );
  }
}
printf("S = singular values (matlab base-1)\n");
printf("U = left singular vectors (matlab base-1)\n");
printf("V = right singular vectors (matlab base-1)\n\n");

...
G.5. 64-bit GESVD

This chapter provides a simple example in the C programming language of how to use cusolverDnXgesvd.

\[ A = U \Sigma V^H \]

A is a 3x2 dense matrix,

\[
A = \begin{pmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0
\end{pmatrix}
\]

The following code uses three steps:

Step 1: compute \( A = U \Sigma V^T \)

Step 2: check accuracy of singular value

Step 3: measure residual \( A - U \Sigma V^T \)

...
Examples of Singular Value Decomposition

```c
const int n = 2;
const int lda = m;
/*
 * A = | 1 2 |
 *     | 4 5 |
 *     | 2 1 |
*/
double A[lda*n] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0};
double U[lda*m]; /* m-by-m unitary matrix */
double VT[lda*n]; /* n-by-n unitary matrix */
double S[n]; /* singular value */
double S_exact[n] = {7.065283497082729, 1.040081297712078};
double *d_A = NULL;
double *d_S = NULL;
double *d_U = NULL;
double *d_VT = NULL;
int *d_info = NULL;
double *d_W = NULL; /* W = S*VT */
void *bufferOnDevice = NULL;
size_t workspaceInBytesOnDevice = 0;
void *bufferOnHost = NULL;
size_t workspaceInBytesOnHost = 0;

int info_gpu = 0;
const double h_one = 1;
const double h_minus_one = -1;
printf("A = (matlab base-1)\n");
printMatrix(m, n, A, lda, "A");
printf("=====
");/* step 1: create cusolverDn/cublas handle */
cusolver_status = cusolverDnCreate(&cusolverH);
assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
cublas_status = cublasCreate(&cublasH);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);
/* step 2: copy A and B to device */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double)*lda*n);
cudaStat2 = cudaMalloc ((void**)&d_S, sizeof(double)*n);
cudaStat3 = cudaMalloc ((void**)&d_U, sizeof(double)*lda*m);
cudaStat4 = cudaMalloc ((void**)&d_VT, sizeof(double)*lda*n);
cudaStat5 = cudaMalloc ((void**)&d_info, sizeof(int));
cudaStat6 = cudaMalloc ((void**)&d_W, sizeof(double)*lda*n);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);
assert(cudaSuccess == cudaStat6);
cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize(); /* wait until d_A is ready */
/* step 3: query working space of SVD */
signed char jobu = 'A'; /* all m columns of U */
signed char jobvt = 'A'; /* all n columns of VT */
cusolver_status = cusolverDnXgesvd_bufferSize (cusolverH,
...
Examples of Singular Value Decomposition

```
NULL, /* params */
jobu,
jobvt,
(int64_t)m,
(int64_t)n,
CUDA_R_64F, /* dataTypeA */
d_A,
(int64_t)lda,
CUDA_R_64F, /* dataTypeS */
d_S,
CUDA_R_64F, /* dataTypeU */
d_U,
(int64_t)lda, /* ldu */
CUDA_R_64F, /* dataTypeVT */
d_VT,
(int64_t)lda, /* ldvt */
CUDA_R_64F, /* computeType */
&workspaceInBytesOnDevice, &workspaceInBytesOnHost);
cudaStat1 = cudaMemcpy((void**)&pBuffer , workspaceInBytes);
assert(cudaSuccess == cudaStat1);
if (0 < workspaceInBytesOnHost){
    bufferOnHost = (void*)malloc(workspaceInBytesOnHost);
    assert(NULL != bufferOnHost);
}
...

/* step 4: compute SVD */
cusolver_status = cusolverDnXgesvd(
cusolverH,
NULL, /* params */
jobu,
jobvt,
(int64_t)m,
(int64_t)n,
CUDA_R_64F, /* dataTypeA */
d_A,
(int64_t)lda,
CUDA_R_64F, /* dataTypeS */
d_S,
CUDA_R_64F, /* dataTypeU */
d_U,
(int64_t)lda, /* ldu */
CUDA_R_64F, /* dataTypeVT */
d_VT,
(int64_t)lda, /* ldvt */
CUDA_R_64F, /* computeType */
bufferOnDevice,
workspaceInBytesOnDevice,
bufferOnHost,
workspaceInBytesOnHost,
d_info);
cudaStat1 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudasuccess == cudaStat1);
cudaStat2 = cudaMemcpy(VT , d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S , d_S, sizeof(double)*n , cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu , d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(VT , d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu , d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(VT , d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu , d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(VT , d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu , d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat1 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(VT , d_VT, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S , d_S, sizeof(double)*n , cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info_gpu , d_info, sizeof(int), cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S , d_S, sizeof(double)*n , cudaMemcpyDeviceToHost);
cudaStat1 = cudaMemcpy(U , d_U , sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
```
Examples of Singular Value Decomposition

```c
printf("after gesvd: info_gpu = %d\n", info_gpu);
assert(0 == info_gpu);
printf("=====\n");

printf("S = (matlab base-1)\n");
printMatrix(n, 1, S, lda, "S");
printf("=====\n");

printf("U = (matlab base-1)\n");
printMatrix(m, m, U, lda, "U");
printf("=====\n");

printf("VT = (matlab base-1)\n");
printMatrix(n, n, VT, lda, "VT");
printf("=====\n");

/* step 5: measure error of singular value */
  double ds_sup = 0;
  for(int j = 0; j < n; j++){
    double err = fabs(S[j] - S_exact[j]);
    ds_sup = (ds_sup > err)? ds_sup : err;
  }
  printf(" |S - S_exact| = %E \n", ds_sup);

/* step 6: |A - U*S*VT| */
  /* W = S*VT */
  cublas_status = cublasDdgmm(
    cublasH,
    CUBLAS_SIDE_LEFT,
    n,
    n,
    d_VT,
    lda,
    d_S,
    l,
    d_W,
    lda);
  assert(CUBLAS_STATUS_SUCCESS == cublas_status);
  /* A := -U*W + A */
  cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
  assert(cudaSuccess == cudaStat1);
  cudaDeviceSynchronize(); /* wait until d_A is ready */
  cublas_status = cublasDgemm_v2(
    cublasH,
    CUBLAS_OP_N,
    CUBLAS_OP_N,
    m, /* number of rows of A */
    n, /* number of columns of A */
    n, /* number of columns of U */
    &h_minus_one,
    d_U,
    lda,
    d_W,
    lda,
    &h_one,
    d_A,
    lda);
  assert(CUBLAS_STATUS_SUCCESS == cublas_status);
  double dR_fro = 0.0;
  cublas_status = cublasDnrm2_v2(
```

G.6. 64-bit GESVDP

This chapter provides a simple example in the C programming language of how to use cusolverDnXgesvdp.

\[ A = U \Sigma V^H \]

A is a 3x2 dense matrix,

\[
A = \begin{pmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0 \\
\end{pmatrix}
\]

The following code uses three steps:
Step 1: compute \( A = U S V^T \)
Step 2: check accuracy of singular value
Step 3: measure residual \( A - U S V^T \)

...
#include <cusolverDn.h>

void printMatrix(int m, int n, const double*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            double Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusolverDnHandle_t cusolverH = NULL;
    cublasHandle_t cublasH = NULL;
    cublasStatus_t cublas_status = CUBLAS_STATUS_SUCCESS;
    cusolverStatus_t cusolver_status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    cudaError_t cudaStat6 = cudaSuccess;
    const int m = 3;
    const int n = 2;
    const int lda = m;
/*
| 1 2 |
| 4 5 |
| 2 1 |
*/
    double A[lda*n] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0};
    double U[lda*m]; /* m-by-m unitary matrix */
    double V[lda*n]; /* n-by-n unitary matrix */
    double S[n]; /* singular value */
    double S_exact[n] = {7.065283497082729, 1.040081297712078};
    double *d_A = NULL;
    double *d_S = NULL;
    double *d_U = NULL;
    double *d_V = NULL;
    int *d_info = NULL;
    double *d_W = NULL; /* W = S*VT */
    void *bufferOnDevice = NULL;
    size_t workspaceInBytesOnDevice = 0;
    void *bufferOnHost = NULL;
    size_t workspaceInBytesOnHost = 0;

    ...
Examples of Singular Value Decomposition

/* step 2: copy A and B to device */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double)*lda*n);
cudaStat2 = cudaMalloc ((void**)&d_S, sizeof(double)*n);
cudaStat3 = cudaMalloc ((void**)&d_U, sizeof(double)*lda*m);
cudaStat4 = cudaMalloc ((void**)&d_V, sizeof(double)*lda*n);
cudaStat5 = cudaMalloc ((void**)&d_info, sizeof(int));
cudaStat6 = cudaMalloc ((void**)&d_W, sizeof(double)*lda*n);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);
assert(cudaSuccess == cudaStat6);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize(); /* wait until d_A is ready */
/* step 3: query working space of SVD */
cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;
const int econ = 0; /* compute 3-by-3 U */
//const int econ = 1; /* compute 3-by-2 U */
double h_err_sigma;

cusolver_status = cusolverDnXgesvdp_bufferSize {
    cusolverH,
    NULL, /* params */
    jobz,
    econ,
    (int64_t)m,
    (int64_t)n,
    CUDA_R_64F, /* dataTypeA */
    d_A, _
    (int64_t)lda,
    CUDA_R_64F, /* dataTypeS */
    d_S,
    CUDA_R_64F, /* dataTypeU */
    d_U, _
    (int64_t)lda, /* ldu */
    CUDA_R_64F, /* dataTypeV */
    d_V, _
    (int64_t)lda, /* ldv */
    CUDA_R_64F, /* computeType */
    &workspaceInBytesOnDevice, 
    &workspaceInBytesOnHost);
    cudaStat1 = cudaMalloc((void**)&bufferOnDevice, workspaceInBytesOnDevice);
assert(cudaSuccess == cudaStat1);
    if (0 < workspaceInBytesOnHost){
        bufferOnHost = (void*)malloc(workspaceInBytesOnHost);
        assert(NULL != bufferOnHost);
    }
}

/* step 4: compute SVD */
cusolver_status = cusolverDnXgesvdp(
    cusolverH,
    NULL, /* params */
    jobz,
    econ,
    (int64_t)m,
    (int64_t)n,
    CUDA_R_64F, /* dataTypeA */
    d_A,
    (int64_t)lda,
    CUDA_R_64F, /* dataTypeS */
Examples of Singular Value Decomposition

```c
  d_S,
  CUDA_R_64F, /* dataTypeU */
  d_U,
  (int64_t)lda, /* ldu */
  CUDA_R_64F, /* dataTypeV */
  d_V,
  (int64_t)lda, /* ldv */
  CUDA_R_64F, /* computeType */
  bufferOnDevice,
  workspaceInBytesOnDevice,
  bufferOnHost,
  workspaceInBytesOnHost,
  d_info,
  &h_err_sigma);
  cudaStat1 = cudaDeviceSynchronize();
  assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
  assert(cudaSuccess == cudaStat1);
  cudaStat1 = cudaMemcpy(U, d_U, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
  cudaStat2 = cudaMemcpy(V, d_V, sizeof(double)*lda*n, cudaMemcpyDeviceToHost);
  cudaStat4 = cudaMemcpy(&info_gpu, d_info, sizeof(int), cudaMemcpyDeviceToHost);
  assert(cudaSuccess == cudaStat2);
  assert(cudaSuccess == cudaStat4);
  cudaDeviceSynchronize(); /* wait until host data is ready */

  printf("after gesvd: info_gpu = %d\n", info_gpu);
  printf("====\n");

  printf("S = (matlab base-1)\n");
  printMatrix(n, 1, S, lda, "S");
  printf("====\n");

  printf("U = (matlab base-1)\n");
  printMatrix(m, (econ)? n:m, U, lda, "U");
  printf("====\n");

  printf("V = (matlab base-1)\n");
  printMatrix(n, n, V, lda, "V");
  printf("====\n");

  /* step 5: measure error of singular value */
  double ds_sup = 0;
  for(int j = 0; j < n; j++)
  {
    double err = fabs( S[j] - S_exact[j] );
    ds_sup = (ds_sup > err)? ds_sup : err;
  }
  printf("|S - S_exact| = %E \n", ds_sup);

  /* step 6: |A - U*S*V**T| */
  /* W = V*S */
  cublas_status = cublasDdgmm(  
    cublasH,
    CUBLAS_SIDE_RIGHT,
    n,
    n,
    d_V,
    lda,
    d_S,
    lda,
    d_W,
  ...
Examples of Singular Value Decomposition

lda);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);
/* A := -U*W**T + A */
cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize(); /* wait until d_A is ready */
cublas_status = cublasDgemm_v2(
cublasH,
CUBLAS_OP_N,
CUBLAS_OP_T,
m, /* number of rows of A */
n, /* number of columns of A */
&h_minus_one,
d_U,
lda,
d_W,
lda,
&h_one,
d_A,
lda);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

double dR_fro = 0.0;
cublas_status = cublasDnrm2_v2(
cublasH, lda*n, d_A, 1, &dR_fro);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

printf("|A - U*S*V**T| = %E \n", dR_fro);
printf("h_err_sigma = %E \n", h_err_sigma);
printf("h_err_sigma is 0 if the singular value of A is not close to zero\n");

/* free resources */
if (d_A    ) cudaFree(d_A);
if (d_S    ) cudaFree(d_S);
if (d_U    ) cudaFree(d_U);
if (d_V    ) cudaFree(d_V);
if (d_info) cudaFree(d_info);
if (d_W    ) cudaFree(d_W);
if (bufferOnDevice) cudaFree(bufferOnDevice);
if (bufferOnHost  ) free(bufferOnHost);
if (cublasH ) cublasDestroy(cublasH);
if (cusolverH) cusolverDnDestroy(cusolverH);
cudaDeviceReset();
return 0;
Appendix H. Examples of multiGPU eigenvalue solver

This chapter provides three examples to perform multiGPU symmetric eigenvalue solver. The difference among them is how to generate the testing matrix. The testing matrix is a tridiagonal matrix, from standard 3-point stencil of Laplacian operator with Dirichlet boundary condition, so each row has \([-1, 2, -1]\) signature.

The spectrum has analytic formula, we can check the accuracy of eigenvalues easily. The user can change the dimension of the matrix to measure the performance of eigenvalue solver.

The example code enables peer-to-peer access to take advantage of NVLINK. The user can check the performance by on/off peer-to-peer access.

The procedures of these three examples are 1) to prepare a tridiagonal matrix in distributed sense, 2) to query size of the workspace and to allocate the workspace for each device, 3) to compute eigenvalues and eigenvectors, and 4) to check accuracy of eigenvalues.

The example 1 allocates distributed matrix by calling `createMat`. It generates the matrix on host memory and copies it to distributed device memory via `memcpyH2D`.

The example 2 allocates distributed matrix manually, generates the matrix on host memory and copies it to distributed device memory manually. This example is for the users who are familiar with data layout of ScaLAPACK.

The example 3 allocates distributed matrix by calling `createMat` and generates the matrix element-by-element on distributed matrix via `memcpyH2D`. The user needs not to know the data layout of ScaLAPACK. It is useful when the matrix is sparse.

H.1. SYEVD of 1D Laplacian operator (example 1)

```c
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 * nvcc -ccbin gcc -I/usr/local/cuda/include -c main.cpp -o main.o
 * nvcc -cudart static main.o -lcusolverMg
 */
#include <stdio.h>
#include <stdlib.h>
```

...


Examples of multiGPU eigenvalue solver

```
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"

//#define SHOW_FORMAT

#ifndef IDX2F
#define IDX2F(i,j,lda) ((((j)-1)*((size_t)lda))+((i)-1))
#endif /* IDX2F */

#ifndef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

static void print_matrix(
    int m,
    int n,
    const double *A,
    int lda,
    const char* name)
{
    printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
    for(int row = 1 ; row <= m ; row++){
        for(int col = 1 ; col <= n ; col++){
            double Aij = A[IDX2F(row, col, lda)];
            printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij );
        }
    }
}

static void gen_1d_laplacian(
    int N,
    double *A,
    int lda)
{
    memset(A, 0, sizeof(double)*lda*N);
    for(int J = 1 ; J <= N; J++ ){
        A[ IDX2F( J, J, lda ) ] = 2.0;
        if( (J-1) >= 1 ){ /* A(J, J-1) = -1*/
            A[ IDX2F( J, J-1, lda ) ] = -1.0;
        }
        if( (J+1) <= N ) {
            A[ IDX2F( J, J+1, lda ) ] = -1.0;
        }
    }
}

...`

```

```
int main( int argc, char* argv[])
{
    cusolverMgHandle_t handle = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat = cudaSuccess;

    /* maximum number of GPUs */
    const int MAX_NUM_DEVICES = 16;

    int nbGpus = 0;
    int deviceList[MAX_NUM_DEVICES];
```
Examples of multiGPU eigenvalue solver

```c
const int N = 2111;
const int IA = 1;
const int JA = 1;
const int TA = 256; /* tile size */
const int lda = N;

double *A = NULL; /* A is N-by-N */
double *D = NULL; /* D is 1-by-N */
int info = 0;

cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;

cudaLibMgMatrixDesc_t descrA;
cudaLibMgGrid_t gridA;
cusolverMgGridMapping_t mapping = CUDALIBMG_GRID_MAPPING_COL_MAJOR;

double **array_d_A = NULL;

int64_t lwork = 0; /* workspace: number of elements per device */
double **array_d_work = NULL;

printf("test 1D Laplacian of order %d\\n", N);

printf("step 1: create Mg handle and select devices \\
");
status = cusolverMgCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat = cudaGetDeviceCount( &nbGpus );
assert( cudaSuccess == cudaStat );

nbGpus = (nbGpus < MAX_NUM_DEVICES)? nbGpus : MAX_NUM_DEVICES;
printf("\nthere are %d GPUs \n", nbGpus);
for(int j = 0 ; j < nbGpus ; j++){
    deviceList[j] = j;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, j);
    printf("device %d, %s, cc %d.%d \\
",j, prop.name, prop.major, prop.minor);
}

status = cusolverMgDeviceSelect(
    handle,
    nbGpus,
    deviceList);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 2: Enable peer access.\\n");
assert( 0 == enablePeerAccess( nbGpus, deviceList ) );

printf("step 3: allocate host memory A \n");
A = (double *)malloc (sizeof(double)*lda*N);
D = (double *)malloc (sizeof(double)*N);
assert( NULL != A );
assert( NULL != D );

printf("step 4: prepare 1D Laplacian \n");
gen_1d_laplacian(
    N,
    &A[ IDX2F( IA, JA, lda ) ],
    lda
);
```

...
examples of multiGPU eigenvalue solver

ifdef SHOW_FORMAT
  print_matrix( N, N, A, lda, "A");
endif

printf("step 5: create matrix descriptors for A and D \n");

status = cusolverMgCreateDeviceGrid(&gridA, 1, nbGpus, deviceList, mapping);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* (global) A is N-by-N */
status = cusolverMgCreateMatrixDesc(
  &descrA,
  N,   /* number of rows of (global) A */
  N,   /* number of columns of (global) A */
  N,   /* number of rows in a tile */
  T_A, /* number of columns in a tile */
  CUDA_R_64F,
  gridA );
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 6: allocate distributed matrices A and D \n");

array_d_A = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_A);

/* A := 0 */
createMat<double>(
  nbGpus,
  deviceList,
  N,   /* number of columns of global A */
  T_A, /* number of columns per column tile */
  lda,  /* leading dimension of local A */
  array_d_A
);

printf("step 7: prepare data on devices \n");
memcpyH2D<double>(
  nbGpus,
  deviceList,
  N,
  N,
  /* input */
  A,
  lda,
  /* output */
  N,   /* number of columns of global A */
  T_A, /* number of columns per column tile */
  lda,  /* leading dimension of local A */
  array_d_A, /* host pointer array of dimension nbGpus */
  IA,
  JA
);

printf("step 8: allocate workspace space \n");
status = cusolverMgSyevd_bufferSize(
  handle,
  (cusolverEigMode_t)jobz,
  CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
  N,
  (void**)array_d_A,
  IA, /* base-1 */
  JA, /* base-1 */
  descrA,
  (void*)D,
  CUDA_R_64F,
  CUDA_R_64F,
&lwork);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("allocate device workspace, lwork = %lld \n", (long long)lwork);
array_d_work = (double**)malloc(sizeof(double)*nbGpus);
assert( NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus,
    deviceList,
    sizeof(double)*lwork, /* number of bytes per device */
    (void**)array_d_work
);
/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);

printf("step 9: compute eigenvalues and eigenvectors \n");
status = cusolverMgSyevd(
    handle,
    (cusolverEigMode_t)jobz,
    CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
    N,
    (void**)array_d_A, /* exit: eigenvectors */
    IA,
    JA,
    descrA,
    (void**)D, /* exit: eigenvalues */
    CUDA_R_64F,
    CUDA_R_64F,
    (void**)array_d_work,
    lwork,
    &info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);
/* check if SYEVD converges */
assert(0 == info);

...

printf("step 10: copy eigenvectors to A and eigenvalues to D\n");

memcpyD2H<double>(
    nbGpus,
    deviceList,
    N,
    N,
    /* input */
    N, /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A,
    IA,
    JA,
    /* output */
    A, /* N-y-N eigenvectors */
    lda
);
Examples of multiGPU eigenvalue solver

```c
#ifdef SHOW_FORMAT
    printf("eigenvalue D = \n");
    /* D is 1-by-N */
    print_matrix(1, N, D, 1, "D");
#endif

    printf("step 11: verify eigenvalues \n");
    printf(" lambda(k) = 4 * sin(pi/2 * k/(N+1))^2 for k = 1:N \n");
    double max_err_D = 0;
    for(int k = 1; k <= N ; k++){
        const double pi = 4*atan(1.0);
        const double h  = 1.0/((double)N+1);
        const double factor = sin(pi/2.0 * ((double)k)*h);
        const double lambda = 4.0*factor*factor;
        const double err = fabs(D[IDX1F(k)] - lambda);
            //        printf("k = %d, D = %E, lambda = %E, err = %E\n", k, D[IDX1F(k)], lambda,
                err);
        max_err_D = (max_err_D > err)? max_err_D : err;
    }
    printf("\n|D - lambda|_inf = %E
\n", max_err_D);

    printf("step 12: free resources \n");
    destroyMat(
        nbGpus,
        deviceList,
        N, /* number of columns of global A */
        T_A, /* number of columns per column tile */
        (void**)array_d_A);
    workspaceFree( nbGpus, deviceList, (void**)array_d_work );
    if (NULL != A) free(A);
    if (NULL != D) free(D);
    if (NULL != array_d_A   ) free(array_d_A);
    if (NULL != array_d_work) free(array_d_work);
    return 0;
}

...  

/* util.hxx */
*/

/* nbGpus : (int) number of gpus in deviceList array.
* deviceList : (*int) list of device ids.
* The function restores the input device before leaving.
*/
static int enablePeerAccess (const int nbGpus, const int *deviceList )
{
    int currentDevice = 0;
    cudaGetDevice( &currentDevice );

    /* Remark: access granted by this cudaDeviceEnablePeerAccess is unidirectional */
```
Examples of multiGPU eigenvalue solver

```c
/* Rows and columns represents a connectivity matrix between GPUs in the system */
for(int row=0; row < nbGpus; row++) {
    cudaSetDevice(row);
    for(int col=0; col < nbGpus; col++) {
        if( row != col ){
            cudaError_t cudaStat1 = cudaSuccess;
            cudaError_t cudaStat2 = cudaSuccess;
            int canAccessPeer = 0;
            cudaStat1 = cudaDeviceCanAccessPeer( &canAccessPeer, row, col );
            if ( canAccessPeer ){
                printf("\t Enable peer access from gpu %d to gpu %d\n", row, col );
            }
            assert(cudaStat1 == cudaSuccess);
            assert(cudaStat2 == cudaSuccess);
        }
    }
    cudaSetDevice( currentDevice );
    return 0;
}

static int workspaceFree(
    int num_devices,
    const int *deviceIdA, /* <int> dimension num_devices */
    void **array_d_work  /* <t> num_devices, host array */
) {
    int currentDev = 0; /* record current device ID */
    cudaGetDevice( &currentDev );
    for(int idx = 0 ; idx < num_devices ; idx++){
        int deviceId = deviceIdA[idx];
        /* WARNING: we need to set device before any runtime API */
        cudaSetDevice( deviceId );
        if (NULL != array_d_work[idx]){  
            cudaFree(array_d_work[idx]);
        }
    }
    cudaSetDevice(currentDev);
    return 0;
}

... static int workspaceAlloc(
    int num_devices,
    const int *deviceIdA, /* <int> dimension num_devices */
    size_t sizeInBytes,  /* number of bytes per device */
    void **array_d_work  /* <t> num_devices, host array */
) {
    cudaError_t cudaStat1 = cudaSuccess;
    int currentDev = 0; /* record current device ID */
    cudaGetDevice( &currentDev );
    size_t totalSize = num_devices * sizeInBytes;
    void *deviceWorkspace = NULL;
    cudaMalloc((void**)&deviceWorkspace, totalSize);
    for(int idx = 0 ; idx < num_devices ; idx++){
        int deviceId = deviceIdA[idx];
        cudaSetDevice( deviceId );
        void *hostWorkspace = NULL;
        cudaMalloc((void**)&hostWorkspace, sizeInBytes);
        cudaMemcpy(hostWorkspace, deviceWorkspace, sizeInBytes, cudaMemcpyDeviceToHost);
        hostWorkspace[
```
Examples of multiGPU eigenvalue solver

```c
memset(array_d_work, 0, sizeof(void*)*num_devices);
for(int idx = 0 ; idx < num_devices ; idx++){
    int deviceId = deviceIdA[idx];
    /* WARNING: we need to set device before any runtime API */
    cudaSetDevice( deviceId );

    void *d_workspace = NULL;
    cudaStat1 = cudaMalloc(&d_workspace, sizeInBytes);
    assert( cudaSuccess == cudaStat1 );
    array_d_work[idx] = d_workspace;
}
 cudaSetDevice(currentDev);
return 0;
}

/* create a empty matrix A with A := 0 */
template <typename T_ELEM>
int createMat(
    int num_devices,
    const int *deviceIdA, /* <int> dimension num_devices */
    int N_A,   /* number of columns of global A */
    int T_A,   /* number of columns per column tile */
    int LLD_A, /* leading dimension of local A */
    T_ELEM **array_d_A  /* host pointer array of dimension num_devices */
)
{
    cudaError_t cudaStat1 = cudaSuccess;
    int currentDev = 0; /* record current device id */
    cudaMemcpy(&currentDev);
    cudaDeviceSynchronize();
    const int A_num_blks = ( N_A + T_A - 1) / T_A;
    const int max_A_num_blks_per_device = (A_num_blks + num_devices-1)/num_devices;
    /* Allocate base pointers */
    memset(array_d_A, 0, sizeof(T_ELEM*) * num_devices);
    for( int p = 0 ; p < num_devices ; p++){
        cudaStat1 = cudaMemcpy(deviceIdA[p]);
        assert(cudaSuccess == cudaStat1);
        /* Allocate max_A_num_blks_per_device blocks per device */
        cudaStat1 =
        cudaMemcpy( &array_d_A[p], sizeof(T_ELEM)*LLD_A*T_A*max_A_num_blks_per_device );
        assert(cudaSuccess == cudaStat1);
        /* A := 0 */
        cudaStat1 = cudaMemcpy( array_d_A[p],
        0, sizeof(T_ELEM)*LLD_A*T_A*max_A_num_blks_per_device );
        assert(cudaSuccess == cudaStat1);
    }
    cudaDeviceSynchronize();
    cudaMemcpy(&currentDev);
    return 0;
}

...

static int destroyMat (
    int num_devices,
    const int *deviceIdA, /* <int> dimension num_devices */
    int N_A,   /* number of columns of global A */
    int T_A,   /* number of columns per column tile */
    void **array_d_A) /* host pointer array of dimension num_devices */
{
    cudaError_t cudaStat1 = cudaSuccess;
    int currentDev = 0; /* record current device id */
    cudaMemcpy(&currentDev);
```
const int num_blocks = (N_A + T_A - 1) / T_A;
for(int p = 0 ; p < num_devices ; p++){
    cudaStat = cudaSetDevice(deviceIdA[p]);
    assert(cudaSuccess == cudaStat);
    if ( NULL != array_d_A[p] ){
        cudaFree( array_d_A[p] );
    }
}
memset(array_d_A, 0, sizeof(void*)*num_devices);
cudaSetDevice(currentDev);
return 0;
}

template <typename T_ELEM>
static int mat_pack2unpack(
    int num_devices,
    int N_A,   /* number of columns of global A */
    int T_A,   /* number of columns per column tile */
    int LLD_A, /* leading dimension of local A */
    T_ELEM **array_d_A_packed,  /* host pointer array of dimension num_devices */
    /* output */
    T_ELEM **array_d_A_unpacked /* host pointer array of dimension num_blks */
)
{
    const int num_blks = ( N_A + T_A - 1) / T_A;
    for(int p_a = 0 ; p_a < num_devices ; p_a++){
        T_ELEM *d_A = array_d_A_packed[p_a];
        int nz_blks = 0;
        for(int JA_blk_id = p_a ; JA_blk_id < num_blks ; JA_blk_id+=num_devices){
            array_d_A_unpacked[JA_blk_id] = d_A + (size_t)LLD_A * T_A * nz_blks ;
            nz_blks++;
        }
    }
    return 0;
}

...
Examples of multiGPU eigenvalue solver

#include <cuda_runtime.h>
#include <cusolverDn.h>
#include <stdlib.h>
#include <stdio.h>

// Quick return if possible */
if ( (0 >= M) || (0 >= N) ){
    return 0;
}

/* consistent checking */
if ( ldb < M ){
    return 1;
}

cudaGetDevice( &currentDev );
cudaDeviceSynchronize();

const int num_blks = ( N_A + T_A - 1) / T_A;

T_ELEM **array_d_A_unpacked = (T_ELEM**)malloc(sizeof(T_ELEM*)*num_blks);
assert(NULL != array_d_A_unpacked);

mat_pack2unpack<T_ELEM>(
    num_devices,
    N_A,   /* number of columns of global A */
    T_A,   /* number of columns per column tile */
    LLD_A, /* leading dimension of local A */
    array_d_A_packed,  /* host pointer array of size num_devices */
    /* output */
    array_d_A_unpacked /* host pointer array of size num_blks */);

/* region of interest is A(IA:IA+N-1, JA:JA+N-1) */
const int N_hat = (JA-1) + N; /* JA is base-1 */

for(int p_a = 0 ; p_a < num_devices ; p_a++){
    /* region of interest: JA_start_blk_id:1:JA_end_blk_id */
    for(int JA_blk_id = p_a; JA_blk_id <= JA_end_blk_id ; JA_blk_id +=num_devices){
        if ( JA_blk_id < JA_start_blk_id ) { continue; }

        const int IBX_A = (1 + JA_blk_id*T_A); /* base-1 */
        const int A_start_col = imax( JA, IBX_A ); /* base-1 */
        const int A_start_row = IA;  /* base-1 */
        const int bdd  = imin( N_hat, (IBX_A + T_A -1) );
        const int IT_A = imin( T_A, (bdd - A_start_col + 1) );

        const int loc_A_start_row = A_start_row;   /* base-1 */
        const int loc_A_start_col = (A_start_col-IBX_A)+1;  /* base-1 */

        T_ELEM *d_A = array_d_A_unpacked[JA_blk_id] + IDX2F( loc_A_start_row,
                        loc_A_start_col, LLD_A );
        const T_ELEM *h_A = h_B + IDX2F( A_start_row - IA + 1, A_start_col - JA + 1, ldb );

        cudaStat1 = cudaMemcpy2D(
            d_A,  /* dst */
            (size_t)LLD_A * sizeof(T_ELEM),
            h_A,   /* src */
            (size_t)LLD_A * sizeof(T_ELEM),
            LLD_A, /* src leading dimension */
            IT_A,  /* src number of columns */
            cudaMemcpyHostToDevice);
    }
}

cusolverDnDestroyArray( &array_d_A_unpacked);

T_ELEM *array_h_A_unpacked = (T_ELEM*)malloc(sizeof(T_ELEM)*N_A);
assert(NULL != array_h_A_unpacked);

mat_unpack2pack<T_ELEM>(
    num_devices,
    N_A,   /* number of columns of global A */
    T_A,   /* number of columns per column tile */
    LLD_A, /* leading dimension of local A */
    array_h_A_unpacked,  /* host pointer array of size num_devices */
    /* output */
    array_d_A_packed /* host pointer array of size num_blks */;

...
Examples of multiGPU eigenvalue solver

```c

h_A, /* src */
(size_t)ldb * sizeof(T_ELEM),
(size_t)M * sizeof(T_ELEM),
(size_t)IT_A,
cudaMemcpyHostToDevice
);
assert( cudaSuccess == cudaStat1 );
}/* for each tile per device */
}/* for each device */
cudaDeviceSynchronize();
cudaSetDevice(currentDev);
if ( NULL != array_d_A_unpacked ) { free(array_d_A_unpacked); }
return 0;
}

/*
 * B(1:M, 1:N) := A(IA:IA+M-1, JA:JA+N-1)
 */
template <typename T_ELEM>
static int memcpyD2H(
    int num_devices,
    const int *deviceIdA, /* <int> dimension num_devices */
    int M,  /* number of rows in local A, B */
    int N,  /* number of columns in local A, B */
    /* input */
    int N_A,  /* number of columns of global A */
    int T_A,  /* number of columns per column tile */
    int LLD_A, /* leading dimension of local A */
    T_ELEM **array_d_A_packed, /* host pointer array of dimension num_devices */
    int IA,  /* base-1 */
    int JA,   /* base-1 */
    /* output */
    T_ELEM *h_B,  /* host array, h_B is M-by-N with leading dimension ldb */
    int ldb
) {
    cudaError_t cudaStat1 = cudaSuccess;
    int currentDev = 0; /* record current device id */

    /* Quick return if possible */
    if ( (0 >= M) || (0 >= N) ){
        return 0;
    }

    /* consistent checking */
    if ( ldb < M ){
        return 1;
    }
    cudaMemcpyDevice( &currentDev );
cudaDeviceSynchronize();

    const int num_blks = ( N_A + T_A - 1) / T_A;
    T_ELEM **array_d_A_unpacked = (T_ELEM**)malloc(sizeof(T_ELEM)*num_blks);
    assert(NULL != array_d_A_unpacked);

    mat_pack2unpack<T_ELEM>(
        num_devices,
        N_A,  /* number of columns of global A */
        T_A,  /* number of columns per column tile */
        LLD_A, /* leading dimension of local A */
        array_d_A_packed,  /* host pointer array of size num_devices */
        array_d_A_unpacked /* host pointer array of size num_blks */
    )
```

...
Examples of multiGPU eigenvalue solver

H.2. SYEVD of 1D Laplacian operator (example 2)

```c
};
/* region of interest is A(IA:IA+N-1, JA:JA+N-1) */
const int N_hat = (JA-1) + N; /* JA is base-1 */
const int JA_start_blk_id = (JA-1)/T_A;
const int JA_end_blk_id   = (N_hat-1)/T_A;
for(int p_a = 0 ; p_a < num_devices ; p_a++){
  /* region of interest: JA_start_blk_id:1:JA_end_blk_id */
  for(int JA_blk_id = p_a; JA_blk_id <= JA_end_blk_id ; JA_blk_id +=num_devices){
    if ( JA_blk_id < JA_start_blk_id ) { continue; }
    /* process column block, A(A_start_row:M_A, A_start_col : (A_start_col + IT_A-1) ) */
    const int IBX_A = (1 + JA_blk_id*T_A); /* base-1 */
    const int A_start_col = imax( JA, IBX_A );   /* base-1 */
    const int A_start_row = IA;  /* base-1 */
    const int bdd  = imin( N_hat, (IBX_A + T_A -1) );
    const int IT_A = imin( T_A, (bdd - A_start_col + 1) );
    const int loc_A_start_row = A_start_row;   /* base-1 */
    const int loc_A_start_col = (A_start_col-IBX_A)+1;  /* base-1 */
    const T_ELEM *d_A = array_d_A_unpacked[JA_blk_id] +
      IDX2F( loc_A_start_row, loc_A_start_col, LLD_A );
    T_ELEM *h_A = h_B + IDX2F( A_start_row - IA + 1, A_start_col - JA + 1,
      ldb );
    cudaStat1 = cudaMemcpy2D(
      h_A, /* dst */
      (size_t)ldb * sizeof(T_ELEM),
      d_A, /* src */
      (size_t)LLD_A * sizeof(T_ELEM),
      (size_t)M * sizeof(T_ELEM),
      (size_t)IT_A,
      cudaMemcpyDeviceToHost
    );
    assert( cudaSuccess == cudaStat1 );
    /* for each tile per device */
  }/* for each device */
cudaDeviceSynchronize();
cudaSetDevice(currentDev);
if ( NULL != array_d_A_unpacked ) { free(array_d_A_unpacked); } 
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"

#include <cuda_runtime.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"
```

H.2. SYEVD of 1D Laplacian operator (example 2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"

#include <cuda_runtime.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"
```

H.2. SYEVD of 1D Laplacian operator (example 2)
Examples of multi-GPU eigenvalue solver

```c
#ifndef IDX2F
#define IDX2F(i,j,lda) ((((j)-1)*((size_t)lda))+((i)-1))
#endif /* IDX2F */

#ifndef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

#define imin(x,y)  (((x) < (y)) ? (x) : (y))

static void print_matrix(
    int m,
    int n,
    const double *A,
    int lda,
    const char* name)
{
    printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
    for(int row = 1 ; row <= m ; row++){
        for(int col = 1 ; col <= n ; col++){
            double Aij = A[IDX2F(row, col, lda)];
            printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij );
        }
    }
}

static void gen_1d_laplacian(
    int N,
    double *A,
    int lda)
{
    memset(A, 0, sizeof(double)*lda*N);
    for(int J = 1 ; J <= N; J++ ){/* A(J,J) = 2 */
        A[ IDX2F( J, J, lda ) ] = 2.0;
        if ( (J-1) >= 1 ){/* A(J, J-1) = -1*/
            A[ IDX2F( J, J-1, lda ) ] = -1.0;
        }
        if ( (J+1) <= N ){/* A(J, J+1) = -1*/
            A[ IDX2F( J, J+1, lda ) ] = -1.0;
        }
    }
}

...

int main( int argc, char* argv[])
{
    cusolverMgHandle_t handle = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat = cudaSuccess;
    /* maximum number of GPUs */
    const int MAX_NUM_DEVICES = 16;

    int nbGpus = 0;
    int deviceList[MAX_NUM_DEVICES];

    const int N   = 2111;
    const int IA  = 1;
    const int JA  = 1;
    const int T_A = 256; /* tile size */
```
const int lda = N;

double *A = NULL; /* A is N-by-N */
double *D = NULL; /* D is 1-by-N */
int info = 0;

cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;

cudaLibMgMatrixDesc_t descrA;
cudaLibMgGrid_t gridA;
cusolverMgGridMapping_t mapping = CUDALIBMG_GRID_MAPPING_COL_MAJOR;

double **array_d_A = NULL;

int64_t lwork = 0; /* workspace: number of elements per device */
double **array_d_work = NULL;

printf("test 1D Laplacian of order %d\n", N);

printf("step 1: create Mg handle and select devices \n");
status = cusolverMgCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat = cudaGetDeviceCount( &nbGpus );
assert( cudaSuccess == cudaStat );

nbGpus = (nbGpus < MAX_NUM_DEVICES)? nbGpus : MAX_NUM_DEVICES;
printf("there are %d GPUs \n", nbGpus);
for(int j = 0 ; j < nbGpus ; j++){
    deviceList[j] = j;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, j);
    printf("device %d, %s, cc %d.%d \n", j, prop.name, prop.major, prop.minor);
}

status = cusolverMgDeviceSelect(
    handle,
    nbGpus,
    deviceList);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 2: Enable peer access \n");
assert( 0 == enablePeerAccess( nbGpus, deviceList ) );

...

printf("step 3: allocate host memory A \n");
A = (double *)malloc (sizeof(double)*lda*N);
D = (double *)malloc (sizeof(double)*N);
assert( NULL != A );
assert( NULL != D );

printf("step 4: prepare 1D Laplacian \n");
gen_1d_laplacian(
    N,
    &A[ IDX2F( IA, JA, lda ) ],
    lda
);

#endif SHOW_FORMAT
    print_matrix( N, N, A, lda, "A");
#endif

printf("step 5: create matrix descriptors for A and D \n");
Examples of multi-GPU eigenvalue solver

```c
status = cusolverMgCreateDeviceGrid(&gridA, 1, nbGpus, deviceList, mapping );
assert(CUSOLVER_STATUS_SUCCESS == status);
/* (global) A is N-by-N */
status = cusolverMgCreateMatrixDesc(
    &descrA,
    N, /* number of rows of (global) A */
    N, /* number of columns of (global) A */
    N, /* number of rows in a tile */
    T_A, /* number of columns in a tile */
    CUDA_R_64F,
    gridA );
assert(CUSOLVER_STATUS_SUCCESS == status);
printf("step 6: allocate distributed matrices A and D \n");
array_d_A = (double**) malloc (sizeof(double*) * nbGpus);
assert( NULL != array_d_A );

const int A_num_blks = ( N + T_A - 1) / T_A;
const int blks_per_device = (A_num_blks + nbGpus-1)/nbGpus;
for( int p = 0 ; p < nbGpus ; p++){
    cudaSetDevice(deviceList[p]);
    cudaStat =
        cudaMalloc( &(array_d_A[p]), sizeof(double)*lda*T_A*blks_per_device );
    assert(cudaSuccess == cudaStat);
}
printf("step 7: prepare data on devices \n");
/* The following setting only works for IA = JA = 1 */
for( int k = 0 ; k < A_num_blks ; k++){ /* k = ibx * nbGpus + p */
    const int p   = (k % nbGpus);
    const int ibx = (k / nbGpus);
    double *h_Ak = A + (size_t)lda*T_A*k;
    double *d_Ak = array_d_A[p] + (size_t)lda*T_A*ibx;
    const int width = imin(T_A, (N - T_A*k) );
    cudaStat = cudaMemcpy(d_Ak, h_Ak, sizeof(double)*lda*width,
        cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat);
    } /* sync all devices */
    cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);

printf("step 8: allocate workspace space \n");
status = cusolverMgSyevd_bufferSize(
    handle,
    (cusolverEigMode_t)jobz,
    CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
    N,
    (void**)array_d_A,
    IA, /* base-1 */
    JA, /* base-1 */
    descrA,
    (void*)D,
    CUDA_R_64F,
    CUDA_R_64F,
    &lwork);
assert(CUSOLVER_STATUS_SUCCESS == status);
printf("allocate device workspace, lwork = %lld \n", (long long)lwork);
```

...
Examples of multiGPU eigenvalue solver

array_d_work = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus,
    deviceList,
    sizeof(double)*lwork, /* number of bytes per device */
    (void**)array_d_work
);

/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);

printf("step 9: compute eigenvalues and eigenvectors \n");
status = cusolverMgSyevd(
    handle,
    (cusolverEigMode_t)jobz,
    CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
    N,
    (void**)array_d_A, /* exit: eigenvectors */
    IA,
    JA,
    descrA,
    (void*)D, /* exit: eigenvalues */
    CUDA_R_64F,
    CUDA_R_64F,
    (void**)array_d_work,
    lwork,
    &info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);

/* check if SYEVD converges */
assert(0 == info);

...  

printf("step 10: copy eigenvectors to A and eigenvalues to D\n");
memcpyp<

#ifdef SHOW_FORMAT
printf("eigenvalue D = \n");
/* D is 1-by-N */
print_matrix(1, N, D, 1, "D");
#endif
#ifndef IDX2F
#define IDX2F(i,j,lda) ((((j)-1)*((size_t)lda))+((i)-1))
#endif /* IDX2F */

H.3. SYEVD of 1D Laplacian operator (example 3)

...
Examples of multiGPU eigenvalue solver

```c
#ifndef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

static void print_matrix(
    int m,
    int n,
    const double *A,
    int lda,
    const char* name)
{
    printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
    for(int row = 1 ; row <= m ; row++){
        for(int col = 1 ; col <= n ; col++){
            double Aij = A[IDX2F(row, col, lda)];
            printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij);
        }
    }
}

/* the caller must set A = 0 */
static void gen_1d_laplacian(
    cusolverMgHandle_t handle,
    int nbGpus,
    const int *deviceList,
    int N,     /* number of columns of global A */
    int T_A,   /* number of columns per column tile */
    int LLD_A, /* leading dimension of local A */
    double **array_d_A /* host pointer array of dimension nbGpus */)
{
    double two = 2.0;
    double minus_one = -1.0;
    for(int J = 1 ; J <= N; J++ ){
        /* A(J,J) = 2 */
        memcpyH2D<double>(nbGpus, deviceList, 1, 1, &two, 1, N, T_A, LLD_A, array_d_A, J, J);
        if ( (J-1) >= 1 ){
            /* A(J, J-1) = -1 */
            memcpyH2D<double>(nbGpus, deviceList, 1, 1, &minus_one, 1, N, T_A, LLD_A, array_d_A, J, J-1);
        }
        if ( (J+1) <= N ){   /* A(J, J+1) = -1 */
            memcpyH2D<double>(nbGpus, deviceList, 1, 1, &minus_one, 1, N, T_A, LLD_A, array_d_A, J, J+1);
        }
    }
}

int main( int argc, char* argv[] )
{
    cusolverMgHandle_t handle = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat = cudaSuccess;

    /* maximum number of GPUs */
    const int MAX_NUM_DEVICES = 16;

    int nbGpus = 0;
    int deviceList[MAX_NUM_DEVICES];

    const int N   = 2111;
    const int IA  = 1;
    const int JA  = 1;
    ...
const int T_A = 256; /* tile size */
const int lda = N;

double *A = NULL; /* A is N-by-N */
double *D = NULL; /* D is 1-by-N */
int info = 0;

cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR;

cudaLibMgMatrixDesc_t descrA;
cudaLibMgGrid_t gridA;
cusolverMgGridMapping_t mapping = CUDALIBMG_GRID_MAPPING_COL_MAJOR;

double **array_d_A = NULL;
int64_t lwork = 0; /* workspace: number of elements per device */
double **array_d_work = NULL;

printf("test 1D Laplacian of order %d\n", N);
printf("step 1: create Mg handle and select devices \n");
status = cusolverMgCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat = cudaGetDeviceCount( &nbGpus );
assert( cudaSuccess == cudaStat );

nbGpus = (nbGpus < MAX_NUM_DEVICES)? nbGpus : MAX_NUM_DEVICES;
printf("\tthere are %d GPUs \n", nbGpus);
for(int j = 0 ; j < nbGpus ; j++){
    deviceList[j] = j;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, j);
    printf("\tdevice %d, %s, cc %d.%d %d\n", j, prop.name, prop.major, prop.minor);
}

status = cusolverMgDeviceSelect(
    handle,
    nbGpus,
    deviceList);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 2: Enable peer access.\n");
assert( 0 == enablePeerAccess( nbGpus, deviceList ) );

...

printf("step 3: allocate host memory A \n");
A = (double *)malloc (sizeof(double)*lda*N);
D = (double *)malloc (sizeof(double)*N);
assert( NULL != A );
assert( NULL != D );

printf("step 4: create matrix descriptors for A and D \n");
status = cusolverMgCreateDeviceGrid(&gridA, 1, nbGpus, deviceList, mapping );
assert(CUSOLVER_STATUS_SUCCESS == status);
/* (global) A is N-by-N */
status = cusolverMgCreateMatrixDesc(
    &descrA,
    N, /* number of rows of (global) A */
    N, /* number of columns of (global) A */
    N, /* number or rows in a tile */
    T_A, /* number of columns in a tile */
    CUDA_R_64F,
Examples of multiGPU eigenvalue solver

gridA);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 5: allocate distributed matrices A and D, A = 0 and D = 0 \n");

array_d_A = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_A);
/* A := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A);

printf("step 6: prepare 1D Laplacian on devices \n");
gen_1d_laplacian(
    handle,
    nbGpus,
    deviceList,
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A);

printf("step 7: allocate workspace space \n");
status = cusolverMgSyevd_bufferSize(
    handle,
    (cusolverEigMode_t)jobz,
    CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
    N,
    (void**)array_d_A,
    IA, /* base-1 */
    JA, /* base-1 */
    descrA,
    (void*)D,
    CUDA_R_64F,
    CUDA_R_64F,
    &lwork);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("allocate device workspace, lwork = %lld \n", (long long)lwork);
array_d_work = (double**)malloc(sizeof(double*)*nbGpus);
assert( NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus,
    deviceList,
    sizeof(double)*lwork, /* number of bytes per device */
    (void**)array_d_work );
/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);

printf("step 8: compute eigenvalues and eigenvectors \n");
status = cusolverMgSyevd(
    handle,
    (cusolverEigMode_t)jobz,
    CUBLAS_FILL_MODE_LOWER, /* only support lower mode */
Examples of multiGPU eigenvalue solver

```
N,
(void**)array_d_A, /* exit: eigenvectors */
IA,
JA,
descrA,
(void*)D, /* exit: eigenvalues */
CUDA_R_64F,
CUDA_R_64F,
(void**)array_d_work,
lwork,
&info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
/* sync all devices */
cudaStat = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat);
/* check if SYEVD converges */
assert(0 == info);
printf("step 9: copy eigenvectors to A and eigenvalues to D\n");
memcpyD2H<double>(
   nbGpus,
   deviceList,
   N,
   N,
   /* input */
   N,  /* number of columns of global A */
   T_A,  /* number of columns per column tile */
   lda,  /* leading dimension of local A */
   array_d_A,
   IA,
   JA,
   /* output */
   A,  /* N-y-N eigenvectors */
   lda
);
#endif SHOW_FORMAT
   printf("eigenvalue D = \n");
   /* D is 1-by-N */
   print_matrix(1, N, D, 1, "D");
#endif
```

```
printf("step 10: verify eigenvalues \n");
printf(" lambda(k) = 4 * sin(pi/2 *k/(N+1))^2 for k = 1:N \n");
double max_err_D = 0;
for(int k = 1; k <= N ; k++){
    const double pi = 4*atan(1.0);
    const double h = 1.0/((double)N+1);
    const double factor = sin(pi/2.0 * ((double)k)*h);
    const double lambda = 4.0*factor*factor;
    const double err = fabs(D[IDX1F(k)] - lambda);
    // printf("k = %d, D = %E, lambda = %E, err = %E\n", k, D[IDX1F(k)], lambda,
    err);
    max_err_D = (max_err_D > err)? max_err_D : err;
}
printf("\n|D - lambda|_inf = %E\n", max_err_D);
```

```
printf("step 11: free resources \n");
destroyMat(
   nbGpus,
   deviceList,
   nGpus,
   deviceList,
   array_d_A,
   lda,
   descrA,
   A,
   lda,
   array_d_work,
   lwork,
   &info /* host */
);
```

```
cuSOLVER Library
```
Examples of multiGPU eigenvalue solver

N, /* number of columns of global A */
T_A, /* number of columns per column tile */
(void**)array_d_A);

workspaceFree( nbGpus, deviceList, (void**)array_d_work );

if (NULL != A) free(A);
if (NULL != D) free(D);

if (NULL != array_d_A ) free(array_d_A);
if (NULL != array_d_work) free(array_d_work);

return 0;
Appendix I. Examples of multiGPU linear solver

This chapter provides examples to perform multiGPU linear solver.
The example code enables peer-to-peer access to take advantage of NVLINK. The user can check the performance by on/off peer-to-peer access.
The example 1 solves linear system by LU with partial pivoting (getrf and getrs). It allocates distributed matrix by calling createMat. Then generates the matrix on host memory and copies it to distributed device memory via memcpyH2D.
The example 1 solves linear system by Cholesky factorization (potrf and potrs). It allocates distributed matrix by calling createMat. Then generates the matrix on host memory and copies it to distributed device memory via memcpyH2D.
The example 1 solves linear system using the inverse of an Hermitian positive definite matrix using (potrf and potri). It allocates distributed matrix by calling createMat. Then generates the matrix on host memory and copies it to distributed device memory via memcpyH2D.

I.1. GETRF and GETRS of 1D Laplacian operator (example 1)

Please refer H.1 for util.hxx.

```c
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 * nvcc -ccbin gcc -I/usr/local/cuda/include -c main.cpp -o main.o
 * nvcc -cudart static main.o -lcusolverMg
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cuda_runtime.h"
#include "util.hxx"

//#define SHOW_FORMAT
#endif /* IDX2F */
```
#ifndef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

static void print_matrix(
    int m,
    int n,
    const double *A,
    int lda,
    const char* name)
{
    printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
    for(int row = 1 ; row <= m ; row++){
        for(int col = 1 ; col <= n ; col++){
            double Aij = A[IDX2F(row, col, lda)];
            printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij);
        }
    }
}

/* compute |x|_inf */
static double vec_nrm_inf(
    int n,
    const double *x)
{
    double max_nrm = 0;
    for(int row = 1; row <= n ; row++){
        double xi = x[IDX1F(row)];
        max_nrm = ( max_nrm > fabs(xi) )? max_nrm : fabs(xi);
    }
    return max_nrm;
}

/* A is 1D laplacian, return A(N:-1:1, :) */
static void gen_1d_laplacian_perm(
    int N,
    double *A,
    int lda)
{
    memset(A, 0, sizeof(double)*lda*N);
    for(int J = 1 ; J <= N; J++ ){
        A[IDX2F( N-J+1, J, lda )] = 2.0;
        if ( (J-1) >= 1 ){
            A[IDX2F( N-J+1, J-1, lda)] = -1.0;
        }
        if ( (J+1) <= N ){
            A[IDX2F( N-J+1, J+1, lda)] = -1.0;
        }
    }
}

int main( int argc, char* argv[])
{
    cusolverMgHandle_t handle = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat = cudaSuccess;
    /* maximum number of GPUs */
    const int MAX_NUM_DEVICES = 16;
    int nbGpus = 0;
    int deviceList[MAX_NUM_DEVICES];
const int N = 611;
const int IA = 1;
const int JA = 1;
const int T_A = 256; /* tile size of A */
const int lda = N;

const int IB = 1;
const int JB = 1;
const int T_B = 100; /* tile size of B */
const int ldb = N;

double *A = NULL; /* A is N-by-N */
double *B = NULL; /* B is N-by-1, right-hand-side vector */
double *X = NULL; /* X is N-by-1, solution vector */
int *IPIV = NULL; /* IPIV is 1-by-N, pivoting sequence */
int info = 0;

cudaLibMgMatrixDesc_t descrA;
cudaLibMgMatrixDesc_t descrB;
cudaLibMgGrid_t gridA;
cudaLibMgGrid_t gridB;
cusolverMgGridMapping_t mapping = CUDALIBMG_GRID_MAPPNG_COL_MAJOR;

dooble **array_d_A = NULL;
dooble **array_d_B = NULL;
int **array_d_IPIV = NULL;

tint64_t lwork_getrf = 0;
tint64_t lwork_getrs = 0;
tint64_t lwork = 0; /* workspace: number of elements per device */
dooble **array_d_work = NULL;

printf("test permuted 1D Laplacian of order %d\n", N);

...
assert( NULL != A );
assert( NULL != B );
assert( NULL != X );
assert( NULL != IPIV );

/* permute 1D Laplacian to enable pivoting */
printf("step 4: prepare permuted 1D Laplacian for A and B = ones(N,1) \n");
gen_1d_laplacian_perm(
    N,
    &A[ IDX2F( IA, JA, lda ) ],
    lda
);
#endif

/* B = ones(N,1) */
for(int row = 1 ; row <= N ; row++){
    B[IDX1F(row)] = 1.0;
}

printf("step 5: create matrix descriptors for A and B \n");
status = cusolverMgCreateDeviceGrid(&gridA, 1, nbGpus, deviceList, mapping );
assert(CUSOLVER_STATUS_SUCCESS == status);
status = cusolverMgCreateDeviceGrid(&gridB, 1, nbGpus, deviceList, mapping );
assert(CUSOLVER_STATUS_SUCCESS == status);

status = cusolverMgCreateMatrixDesc(
    &descrA,
    N, /* number of rows of (global) A */
    N, /* number of columns of (global) A */
    N, /* number of rows in a tile */
    T_A, /* number of columns in a tile */
    CUDA_R_64F,
    gridA );
assert(CUSOLVER_STATUS_SUCCESS == status);

status = cusolverMgCreateMatrixDesc(
    &descrB,
    N, /* number of rows of (global) B */
    1, /* number of columns of (global) B */
    N, /* number of rows in a tile */
    T_B, /* number of columns in a tile */
    CUDA_R_64F,
    gridB );
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 6: allocate distributed matrices A, B and IPIV \n");
array_d_A = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_A);
array_d_B = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_B);
array_d_IPIV = (int**)malloc(sizeof(int*)*nbGpus);
assert(NULL != array_d_IPIV);

/* A := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    N, /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A)
/* B := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    1,   /* number of columns of global B */
    T_B, /* number of columns per column tile */
    ldb, /* leading dimension of local B */
    array_d_B
);

/* IPIV := 0, IPIV is consistent with A */
createMat<int>(
    nbGpus,
    deviceList,
    N,   /* number of columns of global IPIV */
    T_A, /* number of columns per column tile */
    1,   /* leading dimension of local IPIV */
    array_d_IPIV
);

printf("step 7: prepare data on devices \n");
/* distribute A to array_d_A */
memcpyH2D<double>(
    nbGpus,
    deviceList,
    N, /* number of columns of global A */
    lda, /* leading dimension of local A */
    array_d_A, /* host pointer array of dimension nbGpus */
    IA,
    JA
);

/* distribute B to array_d_B */
memcpyH2D<double>(
    nbGpus,
    deviceList,
    N, /* number of columns of global B */
    ldb, /* leading dimension of local B */
    array_d_B, /* host pointer array of dimension nbGpus */
    IB,
    JB
);

printf("step 8: allocate workspace space \n");
status = cusolverMgGetrf_bufferSize(
    handle,
    N,
    (void**)array_d_A,
    IA, /* base-1 */
    JA, /* base-1 */
    descrA,
    array_d_IPIV,
Examples of multiGPU linear solver

```
CUDA_R_64F,
&lwork_getrf);
assert(CUSOLVER_STATUS_SUCCESS == status);

status = cusolverMgGetrs_bufferSize(
    handle,
    CUBLAS_OP_N,
    N,
    1, /* NRHS */
    (void**)array_d_A,
    IA,
    JA,
    descrA,
    array_d_IPIV,
    (void**)array_d_B,
    IB,
    JB,
    descrB,
    CUDA_R_64F,
    &lwork_getrs);
assert(CUSOLVER_STATUS_SUCCESS == status);

lwork = (lwork_getrf > lwork_getrs)? lwork_getrf : lwork_getrs;
printf("\allocate device workspace, lwork = %lld \n", (long long)lwork);
array_d_work = (double**)malloc(sizeof(double)*nbGpus);
assert( NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus, deviceList, sizeof(double)*lwork, /* number of bytes per device */
    (void**)array_d_work
);
    cudaStat = cudaDeviceSynchronize(); /* sync all devices */
    assert(cudaSuccess == cudaStat);
```

```
printf("step 9: solve A*X = B by GETRF and GETRS \n");
status = cusolverMgGetrf(
    handle,
    N,
    N,
    (void**)array_d_A,
    IA,
    JA,
    descrA,
    array_d_IPIV,
    CUDA_R_64F,
    (void**)array_d_work,
    lwork,
    &info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if A is singular */

status = cusolverMgGetrs(
    handle,
    CUBLAS_OP_N,
    N,
    1, /* NRHS */
    (void**)array_d_A,
    IA,
```
cuSOLVER Library

Examples of multiGPU linear solver

JA,
descrA,
array_d_IPIV,
(void**)array_d_B,
IB,
JB,
descrB,
CUDA_R_64F,
(void**)array_d_work,
lwork,
&info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if parameters are valid */
printf("step 10: retrieve IPIV and solution vector X \n");
memcpyD2H<double>(
    nbGpus,
deviceList,
    N,
    1,
    /* input */
    1, /* number of columns of global B */
    T_B, /* number of columns per column tile */
    ldb, /* leading dimension of local B */
    array_d_B,
    IB,
    JB,
    /* output */
    X, /* N-by-1 */
    ldb
);

...  

/* IPIV is consistent with A, use JA and T_A */
memcpyD2H<int>(
    nbGpus,
deviceList,
    1,
    N,
    /* input */
    N, /* number of columns of global IPIV */
    T_A, /* number of columns per column tile */
    1, /* leading dimension of local IPIV */
    array_d_IPIV,
    1,
    JA,
    /* output */
    IPIV, /* 1-by-N */
    1
);

#ifdef SHOW_FORMAT
/* X is N-by-1 */
    print_matrix(N, 1, X, ldb, "X");
#endif

#ifdef SHOW_FORMAT
/* IPIV is 1-by-N */
    printf("IPIV = matlab base-1, 1-by-%d matrix\n", N);
    for(int row = 1; row <= N ; row++){
        ...
Examples of multiGPU linear solver

```

printf("IPIV(%d) = %d \n", row, IPIV[ IDX1F(row) ]);  
}
#endif

printf("step 11: measure residual error |b - A*x| \n");
double max_err = 0;
for(int row = 1; row <= N ; row++){
    double sum = 0.0;
    for(int col = 1; col <= N ; col++){
        double Aij = A[ IDX2F( row, col, lda ) ];
        double xj = X[ IDX1F(col) ];
        sum += Aij*xj;
    }
    double bi = B[ IDX1F(row) ];
    double err = fabs( bi - sum );
    max_err = ( max_err > err )? max_err : err;
}

double x_nrm_inf = vec_nrm_inf(N, X);
double b_nrm_inf = vec_nrm_inf(N, B);;
double A_nrm_inf = 4.0;
double rel_err = max_err/(A_nrm_inf * x_nrm_inf + b_nrm_inf);
printf("\n|b - A*x|_inf = %E\n", max_err);
printf("|x|_inf = %E\n", x_nrm_inf);
printf("|b|_inf = %E\n", b_nrm_inf);
printf("|A|_inf = %E\n", A_nrm_inf);
/* relative error is around machine zero */
/* the user can use |b - A*x|/(N*|A||x|+|b|) as well */
printf("|b - A*x|/(|A||x|+|b|) = %E\n
", rel_err);

...  

printf("step 12: free resources \n");
destroyMat(
    nbGpus,
    deviceList,
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    (void**)array_d_A);

destroyMat(
    nbGpus,
    deviceList,
    1,   /* number of columns of global B */
    T_B, /* number of columns per column tile */
    (void**)array_d_B);

destroyMat(
    nbGpus,
    deviceList,
    N,   /* number of columns of global IPIV */
    T_A, /* number of columns per column tile */
    (void**)array_d_IPIV);

workspaceFree( nbGpus, deviceList, (void**)array_d_work );

if (NULL != A) free(A);
if (NULL != B) free(B);
if (NULL != X) free(X);
if (NULL != IPIV) free(IPIV);
if (NULL != array_d_A ) free(array_d_A);
if (NULL != array_d_B ) free(array_d_B);
if (NULL != array_d_IPIV) free(array_d_IPIV);
if (NULL != array_d_work) free(array_d_work);

return 0;
```
1.2. POTRF and POTRS of 1D Laplacian operator (example 2)

Please refer H.1 for util.hxx.

```c
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *       nvcc -ccbin gcc -I/usr/local/cuda/include -c main.cpp -o main.o
 *       nvcc -cudart static main.o -lcusolverMg
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"

// #define SHOW_FORMAT
#ifndef IDX2F
#define IDX2F(i,j,lda) (((j)-1)*((size_t)lda))+((i)-1))
#endif /* IDX2F */
#ifndef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

static void print_matrix(
  int m,
  int n,
  const double *A,
  int lda,
  const char* name)
{
  printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
  for(int row = 1 ; row <= m ; row++){
    for(int col = 1 ; col <= n ; col++){
      double Aij = A[IDX2F(row, col, lda)];
      printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij );
    }
  }
}

/* compute |x|_inf */
static double vec_nrm_inf(
  int n,
  const double *x)
{
  double max_nrm = 0;
  for(int row = 1; row <= n ; row++){
    double xi = x[ IDX1F(row) ];
    max_nrm = ( max_nrm > fabs(xi) ) ? max_nrm : fabs(xi);
  }
  return max_nrm;
}
```
Examples of multiGPU linear solver

```c
/* A is 1D laplacian, return A(N:-1:1, :) */
static void gen_1d_laplacian(
    int N,
    double *A,
    int lda)
{
    memset(A, 0, sizeof(double)*lda*N);
    for(int J = 1 ; J <= N; J++ ){
        A[ IDX2F( J, J, lda ) ] = 2.0;
        if ( (J-1) >= 1 ){
            A[ IDX2F( J, J-1, lda ) ] = -1.0;
        }
        if ( (J+1) <= N ){
            A[ IDX2F( J, J+1, lda ) ] = -1.0;
        }
    }
}

/* Generate matrix B := A * X */
static void gen_ref_B(
    int N,
    int NRHS,
    double *A,
    int lda,
    double *X,
    int ldx,
    double *B,
    int ldb)
{
    memset(B, 0, sizeof(double)*lda*NRHS);
    for(int J = 1 ; J <= NRHS; J++ ){
        for(int I = 1 ; I <= N; I++ ){
            for(int K = 1 ; K <= N; K++ ){
                double Aik = A[ IDX2F( I, K, lda ) ];
                double Xk = X[ IDX2F( K, J, ldx ) ];
                B[ IDX2F( I, J, ldb ) ] += (Aik * Xk);
            }
        }
    }
}

int main( int argc, char* argv[])
{
    cusolverMgHandle_t handle = NULL;
    cusolverStatus_t status = CUSOLVER_STATUS_SUCCESS;
    cudaError_t cudaStat = cudaSuccess;
    /* maximum local number of GPUs, set by user */
    const int MAX_NUM_DEVICES = 2;
    int nbGpus = 0;
    int deviceList[MAX_NUM_DEVICES];

    const int NRHS = 2;
    const int N    = 8;
    const int IA  = 1;
    const int JA  = 1;
    const int T_A = 256; /* tile size of A */
    const int lda = N;
```
const int IB = 1;
const int JB = 1;
const int T_B = 10; /* tile size of B */
const int ldb = N;

double *A = NULL; /* A is N-by-N */
double *B = NULL; /* B is N-by-NRHS */
double *X = NULL; /* X is N-by-NRHS */
int info = 0;

cudaLibMgMatrixDesc_t descrA;
cudaLibMgMatrixDesc_t descrB;
cudaLibMgGrid_t gridA;
cudaLibMgGrid_t gridB;
cusolverMgGridMapping_t mapping = CUDALIBM_Grid_MAPPING_COL_MAJOR;

double **array_d_A = NULL;
double **array_d_B = NULL;
int64_t lwork_potrf = 0;
int64_t lwork_potrs = 0;
int64_t lwork = 0; /* workspace: number of elements per device */
double **array_d_work = NULL;

printf("test 1D Laplacian of order %d\n", N);
printf("step 1: create Mg handle and select devices \n");
status = cusolverMgCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat = cudaGetDeviceCount( &nbGpus );
assert( cudaSuccess == cudaStat );
nbGpus = (nbGpus < MAX_NUM_DEVICES)? nbGpus : MAX_NUM_DEVICES;
printf("\tthere are %d GPUs \n", nbGpus);
for(int j = 0 ; j < nbGpus ; j++){
    deviceList[j] = j;
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, j);
    printf("\tdevice %d, %s, cc %d.%d \n", j, prop.name, prop.major, prop.minor);
}

status = cusolverMgDeviceSelect(
          handle,
          nbGpus,
          deviceList);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 2: Enable peer access.\n");
assert( 0 == enablePeerAccess( nbGpus, deviceList ) );

printf("step 3: allocate host memory A \n");
A = (double *)malloc (sizeof(double)*lda*N);
B = (double *)malloc (sizeof(double)*ldb*NRHS);
X = (double *)malloc (sizeof(double)*ldb*NRHS);
assert( NULL != A ) ;
assert( NULL != B ) ;
assert( NULL != X ) ;

/* permute 1D Laplacian to enable pivoting */
printf("step 4: prepare 1D Laplacian for A and X = ones(N,NRHS) \n");
gen_1d_laplacian(N,
&A[IDX2F(IA, JA, lda)],
  lda
);
#endif

#ifdef SHOW_FORMAT
  print_matrix( N, N, A, lda, "A");
#endif

/* X = ones(N,1) */
for(int row = 1 ; row <= N ; row++){
  for(int col = 1 ; col <= NRHS ; col++){
    X[IDX2F(row, col, ldb)] = 1.0;
  }
}
#endif

#ifdef SHOW_FORMAT
  print_matrix( N, NRHS, X, ldb, "Reference solution (X)" );
#endif

/* Set B := A * X */
printf( "step 5: create rhs for reference solution on host B = A*X \n");
gen_ref_B ( 
  N,
  NRHS,
  A, /* input */
  lda,
  X, /* input */
  ldb, /* same leading dimension as B */
  B, /* output */
  ldb);
#endif

#ifdef SHOW_FORMAT
  print_matrix( N, NRHS, B, ldb, "Generated rhs (B)" );
#endif

...
array_d_A = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_A);
array_d_B = (double**)malloc(sizeof(double*)*nbGpus);
assert(NULL != array_d_B);

/* A := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    N, /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A
);
/* B := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    NRHS, /* number of columns of global B */
    T_B, /* number of columns per column tile */
    ldb, /* leading dimension of local B */
    array_d_B
);

...
Examples of multiGPU linear solver

CUBLAS_FILL_MODE_LOWER,
N,
(void**)array_d_A,
IA, /* base-1 */
JA, /* base-1 */
descrA,
CUDA_R_64F,
&lwork_potrf);
assert(CUSOLVER_STATUS_SUCCESS == status);

... status = cusolverMgPotrs_bufferSize(
handle,
CUBLAS_FILL_MODE_LOWER,
N,
NRHS, /* number of columns of B */
(void**)array_d_A,
IA,
JA,
descrA,
(void**)array_d_B,
IB,
JB,
descrB,
CUDA_R_64F,
&lwork_potrs);
assert(CUSOLVER_STATUS_SUCCESS == status);
lwork = (lwork_potrf > lwork_potrs) ? lwork_potrf : lwork_potrs;
printf("\tallocation device workspace, lwork = %lld \n", (long long)lwork);

array_d_work = (double**)malloc(sizeof(double)*nbGpus);
assert(NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus,
    deviceList,
    sizeof(double)*lwork, /* number of bytes per device */
    (void**)array_d_work);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
printf("\tstep 10: solve A*X = B by POTRF and POTRS \n");
status = cusolverMgPotrf(
    handle,
    CUBLAS_FILL_MODE_LOWER,
    N,
    (void**)array_d_A,
    IA,
    JA,
descrA,
    CUDA_R_64F,
    (void**)array_d_work,
lwork,
    &info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if A is singular */
Examples of multiGPU linear solver

```c
status = cusolverMgPotrs(
    handle,
    CUBLAS_FILL_MODE_LOWER,
    N,
    NRHS, /* number of columns of B */
    (void**)array_d_A,
    IA,
    JA,
    descrA,
    (void**)array_d_B,
    IB,
    JB,
    descrB,
    CUDA_R_64F,
    (void**)array_d_work,
    lwork,
    &info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if parameters are valid */
printf("step 11: solution vector B 
");
memcpyD2H<double>(
    nbGpus,
    deviceList,
    N,
    NRHS,
    /* input */
    NRHS, /* number of columns of global B */
    T_B, /* number of columns per column tile */
    ldb, /* leading dimension of local B */
    array_d_B,
    IB,
    JB,
    /* output */
    B, /* N-by-1 */
    ldb
);
```

```c
#ifdef SHOW_FORMAT
    /* X is N-by-NRHS */
    print_matrix(N, NRHS, X, ldb, "Reference solution (X)\n");
#endif
printf("step 12: measure residual error |x - x_ref| 
");
double max_err = 0;
for(int col = 1; col <= NRHS ; col++){
    printf("errors for X[\%,\%d] 
", col);
    double err = 0.0; /* absolute error per column */
    for(int row = 1; row <= N ; row++){
        double xij = X[ IDX2F(row, col, ldb) ];
        double bij = B[ IDX2F(row, col, ldb) ];
        double err = fabs(xij - bij);
        max_err = ( err > max_err ) ? err : max_err;
    }
    double x_nrm_inf = vec_nrm_inf(N, &X[ IDX2F( 1, col, ldb) ]);
    double b_nrm_inf = vec_nrm_inf(N, &B[ IDX2F( 1, col, ldb) ]);
    double A_nrm_inf = 4.0;
    double rel_err = max_err/(A_nrm_inf * x_nrm_inf + b_nrm_inf);
```

```
I.3. POTRF and POTRI of 1D Laplacian operator (example 3)

Please refer H.1 for util.hxx.

/*
   How to compile (assume cuda is installed at /usr/local/cuda/)
   nvcc -ccbin gcc -I/usr/local/cuda/include -o main.cpp -o main.o
   nvcc -cudart static main.o -lcusolverMg
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cuda_runtime.h>
#include "cusolverMg.h"
#include "util.hxx"

// #define SHOW_FORMAT

#ifdef IDX2F
#define IDX2F(i,j,lda) ((((j)-1)*((size_t)lda))+((i)-1))
#endif /* IDX2F */

#ifdef IDX1F
#define IDX1F(i) ((i)-1)
#endif /* IDX1F */

printf("\tb - A*x\_inf = %E\n", max_err);
printf("\tx\_inf = %E\n", x_nrm_inf);
printf("\tb\_inf = %E\n", b_nrm_inf);
printf("\tA\_inf = %E\n", A_nrm_inf);
/* relative error is around machine zero */
/* the user can use |b - A*x|/(N*|A|*|x|+|b|) as well */
printf("\t|b - A*x|/(|A|*|x|+|b|) = %E\n", rel_err);
}
printf("step 13: free resources \n");
destroyMat(
    nbGpus,
    deviceList,
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    (void**) array_d_A);
destroyMat(
    nbGpus,
    deviceList,
    1,   /* number of columns of global B */
    T_B, /* number of columns per column tile */
    (void**) array_d_B);
workspaceFree( nbGpus, deviceList, (void**) array_d_work );
Examples of multiGPU linear solver

```c
#include "idxf.h"

static void print_matrix(
    int m,
    int n,
    const double *A,
    int lda,
    const char* name)
{
    printf("%s = matlab base-1, %d-by-%d matrix\n", name, m, n);
    for(int row = 1 ; row <= m ; row++){
        for(int col = 1 ; col <= n ; col++){
            double Aij = A[IDX2F(row, col, lda)];
            printf("%s(%d,%d) = %20.16E\n", name, row, col, Aij);
        }
    }
}

/* compute |x|_inf */
static double vec_nrm_inf(
    int n,
    const double *x)
{
    double max_nrm = 0;
    for(int row = 1; row <= n ; row++){
        double xi = x[IDX1F(row)];
        max_nrm = ( max_nrm > fabs(xi) )? max_nrm : fabs(xi);
    }
    return max_nrm;
}

/* A is 1D laplacian, return A(N:-1:1, :) */
static void gen_1d_laplacian(
    int N,
    double *A,
    int lda)
{
    memset(A, 0, sizeof(double)*lda*N);
    for(int J = 1 ; J <= N; J++ ){
        A[IDX2F(J, J, lda)] = 2.0;
        if ( (J-1) >= 1 )
            A[IDX2F(J, J-1, lda)] = -1.0;
        if ( (J+1) <= N )
            A[IDX2F(J, J+1, lda)] = -1.0;
    }
}

/* Generate matrix B := A * X */
static void gen_ref_B(
    int N,
    int NRHS,
    double *A,
    int lda,
    double *X,
    int ldx,
    double *B,
    int ldb)
{
    memset(B, 0, sizeof(double)*lda*NRHS);
    for(int J = 1 ; J <= NRHS; J++ ){
        for(int I = 1 ; I <= N; I++ ){
            B[IDX2F(I, J, ldb)] = A[IDX2F(I, J, lda)] * X[IDX1F(J)];
        }
    }
}
```

Examples of multiGPU linear solver

```c
for(int K = 1 ; K <= N; K++ ){
    double Aik = A[ IDX2F( I, K, lda ) ];
    double Xk = X[ IDX2F( K, J, ldx ) ];
    B[ IDX2F( I, J, ldb ) ] += (Aik * Xk);
}
}
```
const int NRHS = 2;
const int N = 8;
const int IA = 1;
const int JA = 1;
const int T_A = 256; /* tile size of A */
const int lda = N;
const int IB = 1;
const int JB = 1;
const int T_B = 10; /* tile size of B */
const int ldb = N;
double *A = NULL;    /* A is N-by-N */
double *B = NULL;    /* B is N-by-NRHS */
double *Xref = NULL; /* X is N-by-NRHS */
double *Xans = NULL; /* X is N-by-NRHS */
int info = 0;
cudaLibMgMatrixDesc_t descrA;
cudaLibMgGrid_t gridA;
cusolverMgGridMapping_t mapping = CUDALIBMG_GRID_MAPPING_COL_MAJOR;
double **array_d_A = NULL;
int64_t lwork_potrf = 0;
int64_t lwork_potri = 0;
int64_t lwork = 0; /* workspace: number of elements per device */
double **array_d_work = NULL;
printf("test 1D Laplacian of order %d\n", N);
printf("step 1: create Mg handle and select devices \n");
status = cusolverMgCreate(&handle);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaGetDeviceCount( &nbGpus );
assert( cudaSuccess == cudaStat );
nbGpus = (nbGpus < MAX_NUM_DEVICES)? nbGpus : MAX_NUM_DEVICES;
printf("there are %d GPUs \n", nbGpus);
for(int j = 0 ; j < nbGpus ; j++){
    deviceList[j] = j;
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, j);
    printf("device %d, %s, cc %d.%d \n",j, prop.name, prop.major, prop.minor);
}
status = cusolverMgDeviceSelect(
    handle,
    nbGpus,
    deviceList);
assert(CUSOLVER_STATUS_SUCCESS == status);

printf("step 2: Enable peer access.\n");
assert( 0 == enablePeerAccess( nbGpus, deviceList ) );
printf("step 3: allocate host memory A \n");
A = (double *)malloc (sizeof(double)*lda*N);
B = (double *)malloc (sizeof(double)*ldb*NRHS);
Xref = (double *)malloc (sizeof(double)*ldb*NRHS);
Examples of multiGPU linear solver

Xans = (double *)malloc(sizeof(double)*ldb*NRHS);
assert( NULL != A );
assert( NULL != B );
assert( NULL != Xref );
assert( NULL != Xans );

/* permute 1D Laplacian to enable pivoting */
printf("step 4: prepare 1D Laplacian for A and Xref = ones(N,NRHS) \n");
gen_1d_laplacian(
    N,
    &A[ IDX2F( IA, JA, lda ) ],
    lda
);
#endif

#define SHOW_FORMAT
print_matrix( N, N, A, lda, "A");
#endif

/* X = ones(N,1) */
for(int row = 1 ; row <= N ; row++){
    for(int col = 1 ; col <= NRHS ; col++){
        Xref[IDX2F(row, col, ldb)] = 1.0;
    }
}
#endif

/* Set B := A * X */
printf("step 5: create rhs for reference solution on host B = A*Xref \n");
gen_ref_B ( N,
            NRHS,
            A,    /* input */
            lda,
            Xref, /* input */
            ldb,  /* same leading dimension as B */
            B,    /* output */
            ldb);
#endif

/* Set B := A * X */
printf("step 5: create rhs for reference solution on host B = A*Xref \n");
gen_ref_B ( N,
            NRHS,
            A,    /* input */
            lda,
            Xref, /* input */
            ldb,  /* same leading dimension as B */
            B,    /* output */
            ldb);
#endif

...
Examples of multiGPU linear solver

/* A := 0 */
createMat<double>(
    nbGpus,
    deviceList,
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A
);

printf("step 8: prepare data on devices \n");
/* distribute A to array_d_A */
memcpyH2D<double>(
    nbGpus,
    deviceList,
    N,
    N,
    /* input */
    A,
    lda,
    /* output */
    N,   /* number of columns of global A */
    T_A, /* number of columns per column tile */
    lda, /* leading dimension of local A */
    array_d_A, /* host pointer array of dimension nbGpus */
    IA,
    JA
);

printf("step 9: allocate workspace space \n");
status = cusolverMgPotrf_bufferSize(
    handle,
    CUBLAS_FILL_MODE_LOWER,
    N,
    (void**)array_d_A,
    IA, /* base-1 */
   JA, /* base-1 */
    descrA,
    CUDA_R_64F,
    &lwork_potrf);
assert(CUSOLVER_STATUS_SUCCESS == status);
...

status = cusolverMgPotri_bufferSize(
    handle,
    CUBLAS_FILL_MODE_LOWER,
    N,
    (void**)array_d_A,
    IA,
    JA,
    descrA,
    CUDA_R_64F,
    &lwork_potri);
assert(CUSOLVER_STATUS_SUCCESS == status);

lwork = (lwork_potrf > lwork_potri)? lwork_potrf : lwork_potri;
printf("\allocate device workspace, lwork = %lld \n", (long long)lwork);

array_d_work = (double**)malloc(sizeof(double)*nbGpus);
assert(NULL != array_d_work);
/* array_d_work[j] points to device workspace of device j */
workspaceAlloc(
    nbGpus,
Examples of multiGPU linear solver

deviceList,
sizeof(double)*lwork, /* number of bytes per device */
(void**)&array_d_work
);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);

printf("step 10: solve A*X = B by POTRF and POTRI \n");
status = cusolverMgPotrf(
handle,
CUBLAS_FILL_MODE_LOWER,
N,
(void**)&array_d_A,
IA,
JA,
descrA,
CUDA_R_64F,
(void**)&array_d_work,
lwork,
&info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if A is singular */

... status = cusolverMgPotri(
handle,
CUBLAS_FILL_MODE_LOWER,
N,
(void**)&array_d_A,
IA,
JA,
descrA,
CUDA_R_64F,
(void**)&array_d_work,
lwork,
&info /* host */
);
assert(CUSOLVER_STATUS_SUCCESS == status);
cudaStat = cudaDeviceSynchronize(); /* sync all devices */
assert(cudaSuccess == cudaStat);
assert(0 == info); /* check if parameters are valid */

printf("step 11: Gather inv(A) from devices to host \n");
memcpyD2H<double>(
nbGpus,
deviceList,
N,
N,
/* input */
N, /* number of columns of global B */
T_A, /* number of columns per column tile */
ldb, /* leading dimension of local B */
array_d_A,
IA,
JA,
/* output */
A, /* N-by-1 */
ldb
);
#endif SHOW_FORMAT
Examples of multiGPU linear solver

```c
/* A is N-by-N*/
    print_matrix(N, N, A, lda, "Computed solution inv(A)");
#endif

printf("step 12: solve linear system B := inv(A) * B \n");
solve_system_with_invA(
    N,
    NRHS,
    A,
    lda,
    B,
    ldb,
    Xans,
    ldb);

...  

#ifdef SHOW_FORMAT
    /* A is N-by-N*/
    print_matrix(N, NRHS, Xans, ldb, "Computed solution Xans");
#endif

    printf("step 13: measure residual error \n\n");
    double max_err = 0.0;
    for(int col = 1; col <= NRHS ; col++){
        printf("errors for X[:,%d] \n", col);
        double err = 0.0; /* absolute error per column */
        for(int row = 1; row <= N ; row++){
            double Xref_ij = Xref[ IDX2F(row, col, ldb) ];
            double Xans_ij = Xans[ IDX2F(row, col, ldb) ];
            double err = fabs(Xref_ij - Xans_ij);
            max_err = ( err > max_err ) ? err : max_err;
        }
        double Xref_nrm_inf = vec_nrm_inf(N, &Xref[ IDX2F( 1, col, ldb) ]);
        double Xans_nrm_inf = vec_nrm_inf(N, &Xans[ IDX2F( 1, col, ldb) ]);
        double A_nrm_inf = 4.0;
        double rel_err = max_err/(A_nrm_inf * Xans_nrm_inf + Xref_nrm_inf);
        printf("|b - A*x|_inf = %E, Xref_nrm_inf = %E, Xans_nrm_inf = %E, A_nrm_inf = %E
", rel_err);
        printf("|Xref|_inf = %E, Xref_nrm_inf = %E, Xans_nrm_inf = %E
", rel_err);
        printf("|b - A*x|/(|A|*|x|+|b|) = %E
", rel_err);
    }

    printf("step 14: free resources \n");
    destroyMat(
        nbGpus,
        deviceList,
        N, /* number of columns of global A */
        T_A, /* number of columns per column tile */
        (void**)array_d_A );

    workspaceFree( nbGpus, deviceList, (void**)array_d_work );

    if (NULL != A) free(A);
    if (NULL != B) free(B);
    if (NULL != Xref) free(Xref);
    if (NULL != Xans) free(Xans);
    if (NULL != array_d_A   ) free(array_d_A);
    if (NULL != array_d_work) free(array_d_work);
```
return 0;
}
NVIDIA would like to thank the following individuals and institutions for their contributions:

- CPU LAPACK routines from netlib, CLAPACK-3.2.1 [http://www.netlib.org/clapack/]

The following is license of CLAPACK-3.2.1.

Copyright (c) 1992-2008 The University of Tennessee. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- METIS-5.1.0 [http://glaros.dtc.umn.edu/gkhome/metis/metis/overview]

The following is license of METIS [Apache 2.0 license].

Copyright 1995-2013, Regents of the University of Minnesota

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

- QD (A C++/fortran-90 double-double and quad-double package) [http://crd-legacy.lbl.gov/~dhbailey/mpdist/]

The following is license of QD (modified BSD license).

Copyright (c) 2003-2009, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy) All rights reserved.

1. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

2. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3. You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code (“Enhancements”) to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.
Appendix K. Bibliography


Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks 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

© 2014-2020 NVIDIA Corporation. All rights reserved.