## TABLE OF CONTENTS

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

### Chapter 2. Using the cuSolver API
- 2.1. Thread Safety
- 2.2. Scalar Parameters
- 2.3. Parallelism with Streams
- 2.4. Link Third-party LAPACK Library
- 2.5. convention of info

### Chapter 3. cuSolver Types Reference
- 3.1. cuSolverDN Types
  - 3.1.1. cusolverDnHandle_t
  - 3.1.2. cublasFillMode_t
  - 3.1.3. cublasOperation_t
  - 3.1.4. cusolverEigType_t
  - 3.1.5. cusolverEigMode_t
  - 3.1.6. cusolverStatus_t
- 3.2. cuSolverSP Types
  - 3.2.1. cusolverSpHandle_t
  - 3.2.2. cusparseMatDescr_t
  - 3.2.3. cusolverStatus_t
- 3.3. cuSolverRF Types
  - 3.3.1. cusolverRfHandle_t
  - 3.3.2. cusolverRfMatrixFormat_t
  - 3.3.3. cusolverRfNumericBoostReport_t
  - 3.3.4. cusolverRfResetValuesFastMode_t
  - 3.3.5. cusolverRfFactorization_t
  - 3.3.6. cusolverRfTriangularSolve_t
  - 3.3.7. cusolverRfUnitDiagonal_t
  - 3.3.8. cusolverStatus_t

### Chapter 4. cuSolver Formats Reference
- 4.1. Index Base Format
- 4.2. Vector (Dense) Format
- 4.3. Matrix (Dense) Format
- 4.4. Matrix (CSR) Format
Chapter 5. cuSolverDN: dense LAPACK Function Reference

5.1. cuSolverDN Helper Function Reference

5.1.1. cusolverDnCreate() .......................................................... 17
5.1.2. cusolverDnDestroy() ...................................................... 18
5.1.3. cusolverDnSetStream() .................................................. 18
5.1.4. cusolverDnGetStream() .................................................. 18
5.1.5. cusolverDnCreateSyevjInfo() .......................................... 19
5.1.6. cusolverDnDestroySyevjInfo() ......................................... 19
5.1.7. cusolverDnXsyevjSetTolerance() ....................................... 19
5.1.8. cusolverDnXsyevjSetMaxSweeps() ..................................... 20
5.1.9. cusolverDnXsyevjSetSortEig() .......................................... 20
5.1.10. cusolverDnXsyevjGetResidual() ........................................ 20
5.1.11. cusolverDnXsyevjGetSweeps() ......................................... 21
5.1.12. cusolverDnCreateGesvdjInfo() ........................................ 21
5.1.13. cusolverDnDestroyGesvdjInfo() ....................................... 22
5.1.14. cusolverDnXgesvdjSetTolerance() ..................................... 22
5.1.15. cusolverDnXgesvdjSetMaxSweeps() .................................... 22
5.1.16. cusolverDnXgesvdjSetSortEig() ........................................ 23
5.1.17. cusolverDnXgesvdjGetResidual() ....................................... 23
5.1.18. cusolverDnXgesvdjGetSweeps() ....................................... 23

5.2. Dense Linear Solver Reference

5.2.1. cusolverDn<t>potrf() ...................................................... 25
5.2.2. cusolverDn<t>potrs() ...................................................... 28
5.2.3. cusolverDn<t>potri() ...................................................... 30
5.2.4. cusolverDn<t>getrf() ...................................................... 33
5.2.5. cusolverDn<t>getrs() ...................................................... 36
5.2.6. cusolverDn<t>geqrf() ...................................................... 38
5.2.7. cusolverDn<t>ormqr() ...................................................... 41
5.2.8. cusolverDn<t>orgqr() ...................................................... 45
5.2.9. cusolverDn<t>sytrf() ...................................................... 48
5.2.10. cusolverDn<t>potrfBatched() .......................................... 51
5.2.11. cusolverDn<t>potrsBatched() .......................................... 54

5.3. Dense Eigenvalue Solver Reference

5.3.1. cusolverDn<t>gebrd() ...................................................... 56
5.3.2. cusolverDn<t>orgbr() ...................................................... 60
5.3.3. cusolverDn<t>sytrd() ...................................................... 64
5.3.4. cusolverDn<t>ormtr() ...................................................... 68
5.3.5. cusolverDn<t>orgtr() ...................................................... 72
5.3.6. cusolverDn<t>gesvd() ...................................................... 75
5.3.7. cusolverDn<t>gesvdj() ..................................................... 80
5.3.8. cusolverDn<t>gesvdjBatched() .......................................... 85
5.3.9. cusolverDn<t>gesvdaStridedBatched() ................................ 90
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.10</td>
<td>cusolverDn&lt;t&gt;syevd()</td>
<td>96</td>
</tr>
<tr>
<td>5.3.11</td>
<td>cusolverDn&lt;t&gt;syevdx()</td>
<td>100</td>
</tr>
<tr>
<td>5.3.12</td>
<td>cusolverDn&lt;t&gt;syevd()</td>
<td>106</td>
</tr>
<tr>
<td>5.3.13</td>
<td>cusolverDn&lt;t&gt;syevdx()</td>
<td>111</td>
</tr>
<tr>
<td>5.3.14</td>
<td>cusolverDn&lt;t&gt;syevj()</td>
<td>117</td>
</tr>
<tr>
<td>5.3.15</td>
<td>cusolverDn&lt;t&gt;sygvj()</td>
<td>122</td>
</tr>
<tr>
<td>5.3.16</td>
<td>cusolverDn&lt;t&gt;syevjBatched()</td>
<td>128</td>
</tr>
</tbody>
</table>

Chapter 6. cuSolverSP: sparse LAPACK Function Reference .................................................. 133

6.1. Helper Function Reference ...........................................................................133
   6.1.1. cusolverSpCreate() ............................................................................... 133
   6.1.2. cusolverSpDestroy() ............................................................................. 133
   6.1.3. cusolverSpSetStream() ........................................................................... 134
   6.1.4. cusolverSpXcsrissym() ............................................................................ 134

6.2. High Level Function Reference ...................................................................... 135
   6.2.1. cusolverSp<t>csrlsvlu() ........................................................................ 136
   6.2.2. cusolverSp<t>csrlsvqr() ....................................................................... 139
   6.2.3. cusolverSp<t>csrlsvchol() ..................................................................... 142
   6.2.4. cusolverSp<t>csrlsqvqr() ...................................................................... 145
   6.2.5. cusolverSp<t>csreigvsil() ..................................................................... 149
   6.2.6. cusolverSp<t>csreigs() .......................................................................... 153

6.3. Low Level Function Reference ...................................................................... 155
   6.3.1. cusolverSpXcsrsymrcm() ....................................................................... 155
   6.3.2. cusolverSpXcsrsymmdq() ...................................................................... 157
   6.3.3. cusolverSpXcsrsymamd() ...................................................................... 158
   6.3.4. cusolverSpXcsrmetisnd() ...................................................................... 160
   6.3.5. cusolverSpXcsrzfd() ............................................................................ 162
   6.3.6. cusolverSpXcsrperm() .......................................................................... 164
   6.3.7. cusolverSpXcsrsqrBatched() .................................................................. 166

Chapter 7. cuSolverRF: Refactorization Reference ....................................................175

7.1. cusolverRfAccessBundledFactors() ................................................................. 175
7.2. cusolverRfAnalyze() .................................................................................... 176
7.3. cusolverRfSetupDevice() .............................................................................. 177
7.4. cusolverRfSetupHost() ................................................................................. 179
7.5. cusolverRfCreate() ..................................................................................... 181
7.6. cusolverRfExtractBundledFactorsHost() ........................................................ 182
7.7. cusolverRfExtractSplitFactorsHost() .............................................................. 183
7.8. cusolverRfDestroy() .................................................................................... 184
7.9. cusolverRfGetMatrixFormat() ...................................................................... 184
7.10. cusolverRfGetNumericProperties() ............................................................... 185
7.11. cusolverRfGetNumericBoostReport() ............................................................ 185
7.12. cusolverRfGetResetValuesFastMode() ........................................................... 186
7.13. cusolverRfGet_Algs() ............................................................................... 186
7.14. cusolverRfRefactor() ................................................................................. 186
Chapter 1.
INTRODUCTION

The cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries. It combines three separate libraries under a single umbrella, each of which can be used independently or in concert with other toolkit libraries.

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.

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()`, `cudaFree()`, `cudaMemcpy()`, and `cudaMemcpyAsync()`.

The cuSolver library requires hardware with a CUDA compute capability (CC) of at least 2.0 or higher. Please see the NVIDIA CUDA C Programming Guide, Appendix A 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 a sparse linear system

\[ Ax = b \]

and the least-squares problem

\[ x = \text{argmin} \| A^* z - b \| \]

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, i.e. 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

\[ A_i x_i = f_i \]

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 functions are available for data types float, double, cuComplex, and cuDoubleComplex. The naming convention 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 (potrf), LU with partial pivoting (getrf), QR factorization (geqrf) and Bunch-Kaufman factorization (sytrf).

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 ls, lsq, eig, 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>csrlsvlu</td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>LU (lu) with partial pivoting</td>
</tr>
<tr>
<td>csrlsvqr</td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>QR factorization (qr)</td>
</tr>
<tr>
<td>csrlsvchol</td>
<td>csr</td>
<td>linear solver (ls)</td>
<td>vector (v)</td>
<td>Cholesky factorization (chol)</td>
</tr>
<tr>
<td>csrlsqvqr</td>
<td>csr</td>
<td>least-square solver (lsq)</td>
<td>vector (v)</td>
<td>QR factorization (qr)</td>
</tr>
<tr>
<td>csreigvsi</td>
<td>csr</td>
<td>eigenvalue solver (eig)</td>
<td>vector (v)</td>
<td>shift-inverse</td>
</tr>
<tr>
<td>csreigs</td>
<td>csr</td>
<td>number of eigenvalues in a box (eigs)</td>
<td>vector (v)</td>
<td></td>
</tr>
<tr>
<td>csrsymrcm</td>
<td>csr</td>
<td>Symmetric Reverse Cuthill-McKee (symrcm)</td>
<td>vector (v)</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 `cudaDeviceSynchronize()` 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. (i.e. 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

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

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

2.3. 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.
2.4. Link Third-party LAPACK Library

Starting with CUDA 10.0, you can link your application with libcusolver_static.a, and choose to use a LAPACK library of your choice. You can use a third-party LAPACK library, for example, MKL, or you can use NVIDIA LAPACK library liblapack_static.a.

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 a LAPACK library, otherwise the linker will report missing symbols. If you use a third-party LAPACK library, you may take the advantage of the third-party LAPACK library routines to accelerate symmetric eigenvalue solver and singular value decomposition.
- 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.

2.5. 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.
### 3.1. 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.

#### 3.1.1. 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.

#### 3.1.2. 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>

#### 3.1.3. 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.
3.1.4. cusolverEigType_t

The `cusolverEigType_t` type indicates which type of eigenvalue solver is. Its values correspond to Fortran integer 1 ($A\times x = \lambda B\times x$), 2 ($A^tB\times x = \lambda x$), 3 ($B^tA\times x = \lambda x$), 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>$A\times x = \lambda B\times x$</td>
</tr>
<tr>
<td>CUSOLVER_EIG_TYPE_2</td>
<td>$A^tB\times x = \lambda x$</td>
</tr>
<tr>
<td>CUSOLVER_EIG_TYPE_3</td>
<td>$B^tA\times x = \lambda x$</td>
</tr>
</tbody>
</table>

3.1.5. 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>
<tbody>
<tr>
<td>CUSOLVER_EIG_MODE_NOVECTOR</td>
<td>only eigenvalues are computed</td>
</tr>
<tr>
<td>CUSOLVER_EIG_MODEVECTOR</td>
<td>both eigenvalues and eigenvectors are computed</td>
</tr>
</tbody>
</table>

3.1.6. cusolverStatus_t

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

3.2. 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`.

3.2.1. 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.
3.2.2. 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`.

3.2.3. cusolverStatus_t

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

<table>
<thead>
<tr>
<th>cusolverStatus_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
</tbody>
</table>
| CUSOLVER_STATUS_NOT_INIT          | 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.  
**To correct:** 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. |
| CUSOLVER_STATUS_ALLOC_FAILED      | Resource allocation failed inside the cuSolver library. This is usually caused by a `cudaMalloc()` failure.  
**To correct:** prior to the function call, deallocate previously allocated memory as much as possible. |
| CUSOLVER_STATUS_INVALID_VALUE     | An unsupported value or parameter was passed to the function (a negative vector size, for example).  
**To correct:** ensure that all the parameters being passed have valid values. |
| CUSOLVER_STATUS_ARCH_MISMATCH     | 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
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.

3.3. cuSolverRF Types

cuSolverRF only supports double.

3.3.1. 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.

3.3.2. cusolverRfMatrixFormat_t

The cusolverRfMatrixFormat_t is an enum that indicates the input/output matrix format assumed by the cusolverRfSetupDevice(), cusolverRfSetupHost(), cusolverRfResetValues(), cusolverRfExtractBundledFactorsHost() 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>

3.3.3. 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>
3.3.4. 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>

3.3.5. 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>

3.3.6. 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_ALG0</td>
<td>algorithm 0.</td>
</tr>
<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>

3.3.7. 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>
<tbody>
<tr>
<td>CUSOLVER_UNIT_DIAGONAL_STORED_L</td>
<td>unit diagonal is stored in lower triangular factor. (default)</td>
</tr>
<tr>
<td>CUSOLVER_UNIT_DIAGONAL_STORED_U</td>
<td>unit diagonal is stored in upper triangular factor.</td>
</tr>
<tr>
<td>CUSOLVER_UNIT_DIAGONAL_ASSUMED_L</td>
<td>unit diagonal is assumed in lower triangular factor.</td>
</tr>
<tr>
<td>CUSOLVER_UNIT_DIAGONAL_ASSUMED_U</td>
<td>unit diagonal is assumed in upper triangular factor.</td>
</tr>
</tbody>
</table>
3.3.8. 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.
Chapter 4.
CUSOLVER FORMATS REFERENCE

4.1. 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.

4.2. Vector (Dense) Format
The vectors are assumed to be stored linearly in memory. For example, the vector

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

is represented as

$$(x_1 \; x_2 \; \ldots \; x_n)$$

4.3. 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 $m \times n$ (sub-)matrix

$$\begin{bmatrix} a_{1,1} & \ldots & a_{1,n} \\ a_{2,1} & \ldots & a_{2,n} \\ \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{bmatrix}$$

is represented as
with its elements arranged linearly in memory as

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

where \( l_{da} \geq m \) is the leading dimension of \( A \).

### 4.4. 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></td>
<td>the number of rows (and columns) in the matrix.</td>
</tr>
<tr>
<td>( \text{nnz} )</td>
<td>(int)</td>
<td></td>
<td>the number of non-zero elements in the matrix.</td>
</tr>
<tr>
<td>( \text{csrRowPtr} )</td>
<td>(int *)</td>
<td>( n+1 )</td>
<td>the array of offsets corresponding to the start of each row in the arrays ( \text{csrColInd} ) and ( \text{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>( \text{csrColInd} )</td>
<td>(int *)</td>
<td>( \text{nnz} )</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.</td>
</tr>
<tr>
<td>( \text{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

\[
\text{csrRowPtr} = (0 \ 2 \ 4 \ 8 \ 9)
\]
4.5. 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 <code>cscRowInd</code> and <code>cscVal</code>. This array has also an extra entry at</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the 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>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>
</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

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

is represented as

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

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

\[
cscVal = (1.0 \ 2.0 \ 3.0 \ 4.0 \ 5.0 \ 6.0 \ 7.0 \ 8.0 \ 9.0)
\]
This chapter describes the API of cuSolverDN, which provides a subset of dense LAPACK functions.

5.1. cuSolverDN Helper Function Reference

The cuSolverDN helper functions are described in this section.

5.1.1. cusolverDnCreate()

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. |
5.1.2. cusolverDnDestroy()

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**

<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>

5.1.3. 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**

<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>

5.1.4. 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**

<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>

www.nvidia.com
cuSOLVER Library
5.1.5. 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

<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>

5.1.6. 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. |

5.1.7. 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. |
5.1.8. 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.

5.1.9. cusolverDnXsyevjSetSortEig()

cusolverStatus_t
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.

5.1.10. 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.
## 5.1.11. cuSolverDnXsyevjGetSweeps()

```c
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>
<tr>
<td>info</td>
<td>host</td>
<td>input</td>
<td>the pointer to the structure of syevj.</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>Description</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>

## 5.1.12. 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

<table>
<thead>
<tr>
<th>Status Returned</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>
5.1.13. cusolverDnDestroyGesvdjInfo()

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.

5.1.14. cusolverDnXgesvdjSetTolerance()

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>

Status Returned

CUSOLVER_STATUS_SUCCESS the operation completed successfully.

5.1.15. cusolverDnXgesvdjSetMaxSweeps()

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 gesvdj.</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.
5.1.16. cusolverDnXgesvdjSetSortEig()

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 gesvdj.</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. |

5.1.17. cusolverDnXgesvdjGetResidual()

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>
<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>residual</td>
<td>host</td>
<td>output</td>
<td>residual of gesvdj.</td>
</tr>
</tbody>
</table>

Status Returned

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

5.1.18. 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>

5.2. Dense Linear Solver Reference

This chapter describes linear solver API of cuSolverDN, including Cholesky factorization, LU with partial pivoting, QR factorization and Bunch-Kaufman (LDLT) factorization.
5.2.1. 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);

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

cusolverStatus_t
cusolverDnCpotrf_bufferSize(cusolverDnHandle_t handle,
                            _cublasFillMode_t uplo,
                            int n,
                            cuComplex *A,
                            int lda,
                            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.

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×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 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>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>------</td>
<td>------</td>
<td>-------</td>
<td>------------------------------------------------------------------------------------------------</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 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>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 potrf_bufferSize.</td>
</tr>
<tr>
<td>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the Cholesky factorization is successful. if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo = 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 (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>
5.2.2. 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);

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

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

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

\[ A^t X = B \]

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 \( L \) correospoding to
If input parameter `uplo` is `CUBLAS_FILL_MODE_UPPER`, \( A \) is upper triangular Cholesky factor \( U \) corresponding to \( A = U^H \times U \).

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

If output parameter `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>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>( A )</td>
<td>device</td>
<td>input</td>
<td>(&lt;\text{type}&gt;) 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>( B )</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) 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>devInfo</td>
<td>device</td>
<td>output</td>
<td>if devInfo = 0, the Cholesky factorization 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 (( n \leq 0 ), ( nrhs \leq 0 ), ( lda \leq \max(1, n) ) or ( ldb \leq \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>
5.2.3. cusolverDn<t>potri()

These helper functions calculate the necessary size of work buffers.

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 );

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

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

```c

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

cusolverStatus_t
cusolverDnZpotri(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

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

computed by `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 `uplo` indicates which part of the matrix is used. The function would leave the other part untouched.

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

If the input parameter `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 `Workspace`. The input parameter `Lwork` is the size of the working space, returned by `potri_bufferSize()`.

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

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

**API of 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>Parameter</td>
<td>Host/Device</td>
<td>Input/Output</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------</td>
<td>-------------</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 lda * n 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 (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>
5.2.4. cusolverDn<t>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 );
```

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

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

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

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

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

```c
 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.

```c
 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

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

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.

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` 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.

**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 <code>lda * n</code> with ( lda ) is not less than ( \max(1,m) ).</td>
</tr>
</tbody>
</table>
**lda** | **host** | **input** | leading dimension of two-dimensional array used to store matrix A.
--- | --- | --- | 
**Workspace** | **device** | **in/out** | working space, <type> array of size Lwork.
--- | --- | --- | 
**devIpiv** | **device** | **output** | array of size at least min(m,n), containing pivot indices.
--- | --- | --- | 
**devInfo** | **device** | **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 U(i,i) = 0.

**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>
5.2.5. cusolverDn<t>getrs()

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);

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);

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);

cusolverStatus_t
cusolverDnZgetrs(cusolverDnHandle_t handle,
    cublasOperation_t trans,
    int n,
    int nrhs,
    const cuDoubleComplex *A,
    int lda,
    const int *devIpiv,
    cuDoubleComplex *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 `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 `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 `devIpiv` is an output of `getrf`. It contains pivot indices, which are used to permute right-hand sides.

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

The user can combine `getrf` and `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>$&lt;\text{type}&gt;$ 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>$&lt;\text{type}&gt;$ 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 $devInfo = 0$, the operation 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</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_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>
5.2.6. 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 );

cusolverStatus_t
cusolverDnDgeqrf_bufferSize(cusolverDnHandle_t handle,
   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.

```c
 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,
    int m,
    int n,
    cuDoubleComplex *A,
    int lda,
    cuDoubleComplex *TAU,
    cuDoubleComplex *Workspace,
    int Lwork,
    int *devInfo);

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

$$I - 2v^*vH = I - \tau q^*qH$$

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>&lt;type&gt; array of dimension lda * n with lda is not less than max(1,m).</td>
</tr>
</tbody>
</table>
### lda
- **host**: input
- **Purpose**: leading dimension of two-dimensional array used to store matrix A.

### TAU
- **device**: output
- **Purpose**: <type> array of dimension at least \( \min(m,n) \).

### Workspace
- **device**: in/out
- **Purpose**: working space, <type> array of size Lwork.

### Lwork
- **host**: input
- **Purpose**: size of working array Workspace.

### devInfo
- **device**: output
- **Purpose**: if \( \text{devInfo} = 0 \), the LU factorization is successful. If \( \text{devInfo} = -i \), the \( i \)-th parameter is wrong (not counting handle).

### 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 \( (m,n<0 \text{ or } \text{lda} < \max(1,m)) \).
- **CUSOLVER_STATUS_ARCH_MISMATCH**: the device only supports compute capability 2.0 and above.
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.
5.2.7. cusolverDn<t>ormqr()

These helper functions calculate the size of work buffers needed.

```c

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 *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 *C,  
int ldc,  
int *lwork);

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

cusolverStatus_t
cusolverDnZunmqr_bufferSize(  
cusolverDnHandle_t handle,  
cublasSideMode_t side,  
cublasOperation_t trans,  
int m,  
int n,  
int k,  
const cuDoubleComplex *A,  
int lda,  
const cuDoubleComplex *C,  
int ldc,  
int *lwork);
```

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 lwork,
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,
int lwork,
int *devInfo);
The C and Z data types are complex valued single and double precision, respectively.

The operation of $Q$ is defined by

$$
\text{op}(Q) = \begin{cases} 
Q & \text{if transa} == \text{CUBLAS}_\text{OP}_\text{N} \\
Q^T & \text{if transa} == \text{CUBLAS}_\text{OP}_\text{T} \\
Q^H & \text{if transa} == \text{CUBLAS}_\text{OP}_\text{C} 
\end{cases}
$$

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

$Q = H(1) \ H(2) \ ... \ H(k)$

$Q$ is of order $m$ if $\text{side} = \text{CUBLAS}_\text{SIDE}_\text{LEFT}$ and of order $n$ if $\text{side} = \text{CUBLAS}_\text{SIDE}_\text{RIGHT}$.

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 $\text{geqrf_bufferSize()}$ or $\text{ormqr_bufferSize()}$. 

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 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 lda * k with lda is not less than max(1,m). The matrix A is from 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 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 at least min(m,n). The vector tau is from 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 ldc * n. On exit, C is overwritten by op(Q) * C.</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>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 wrong lda or ldc).</td>
</tr>
</tbody>
</table>
### 5.2.8. `cusolverDn<t>orgqr()`

These helper functions calculate the size of work buffers needed.

```c
kusolverStatus_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.

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

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

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

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

cusolverStatus_t
cusolverDnZungqr(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int k,
    cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *tau,
    cuDoubleComplex *work,
    int lwork,
    int *devInfo);

This function overwrites \( m \times n \) matrix \( A \) by
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 $\text{orgqr_bufferSize()}$.

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

The user can combine $\text{geqrf, orgqr}$ to complete orthogonalization. Please refer to appendix C.2.

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

<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>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix $Q$. $m \geq 0$;</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix $Q$. $m \geq n \geq 0$;</td>
</tr>
<tr>
<td>k</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>$A$</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 $\max(1,m)$. i-th column of $A$ 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 $A$. $lda \geq \max(1,m)$.</td>
</tr>
<tr>
<td>$tau$</td>
<td>device</td>
<td>output</td>
<td>$&lt;\text{type}&gt;$ array of dimension $k$. $tau(i)$ 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, $&lt;\text{type}&gt;$ 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>$\text{devInfo}$</td>
<td>device</td>
<td>output</td>
<td>if $info = 0$, the orgqr is successful. if $info = -i$, 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 ($m,n,k&lt;0$, $n&gt;m$, $k&gt;n$ 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>
</tbody>
</table>
CUSOLVER_STATUS_INTERNAL_ERROR | an internal operation failed.

5.2.9. cusolverDn<t>sytrf()

These helper functions calculate the size of the needed buffers.

```c
void cusolverDnSsymv(cusolverDnHandle_t handle, int n, float *A, int lda, float *work);

void cusolverDnDsymv(cusolverDnHandle_t handle, int n, double *A, int lda, double *work);

void cusolverDnCsymv(cusolverDnHandle_t handle, int n, cuComplex *A, int lda, cuComplex *work);

void cusolverDnZsymv(cusolverDnHandle_t handle, int n, cuDoubleComplex *A, int lda, cuDoubleComplex *work);
```

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

```c
void cusolverDnSgetrf(cusolverDnHandle_t handle, int *info, int *Lwork);

void cusolverDnDgetrf(cusolverDnHandle_t handle, int *info, int *Lwork);
```

```c
void cusolverDnSgetri(cusolverDnHandle_t handle, const int *info, float *A, int lda, float *work);

void cusolverDnDgetri(cusolverDnHandle_t handle, const int *info, double *A, int lda, double *work);
```

```c
void cusolverDnCgetri(cusolverDnHandle_t handle, const int *info, cuComplex *A, int lda, cuComplex *work);

void cusolverDnZgetri(cusolverDnHandle_t handle, const int *info, cuDoubleComplex *A, int lda, cuDoubleComplex *work);
```
The C and Z data types are complex valued single and double precision, respectively.

```c

cusolverStatus_t
cusolverDnCsytrf(cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    int *ipiv,
    cuComplex *work,
    int lwork,
    int *devInfo );

cusolverStatus_t
cusolverDnZsytrf(cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    int *ipiv,
    cuDoubleComplex *work,
    int lwork,
    int *devInfo );
```

This function computes the Bunch-Kaufman factorization of a \( n \times n \) symmetric indefinite matrix

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

If input parameter \( \text{uplo} \) is 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.

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

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

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

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 CUBLAS_FILL_MODE_UPPER and \( \text{devIpiv}(i-1) = \text{devIpiv}(i) = -m < 0 \), \( D(i-1:i,i-1:i) \) is a 2x2 block, and \( (i-1) \)-\( \text{th} \) row/column is interchanged...
with $m$-th row/column. If `uplo` is `CUBLAS_FILL_MODE_LOWER` and `devIpiv(i+1) = 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.

**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><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 $lda$ is not less than $max(1,n)$.</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>ipiv</code></td>
<td>device</td>
<td>output</td>
<td>array of size at least $n$, containing pivot indices.</td>
</tr>
<tr>
<td><code>work</code></td>
<td>device</td>
<td>in/out</td>
<td>working space, $&lt;$type$&gt;$ array of size <code>lwork</code>.</td>
</tr>
<tr>
<td><code>lwork</code></td>
<td>host</td>
<td>input</td>
<td>size of working space <code>work</code>.</td>
</tr>
<tr>
<td><code>devInfo</code></td>
<td>device</td>
<td>output</td>
<td>if <code>devInfo = 0</code>, the LU factorization is successful. If <code>devInfo = -i</code>, the $i$-th parameter is wrong (not counting handle). If <code>devInfo = i</code>, $D(i,i) = 0$.</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 ($n&lt;0$ 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>
5.2.10. cusolverDn\textless{}t\textgreater{}potrfBatched()

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


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

cusolverStatus_t
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.


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

cusolverStatus_t
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 \texttt{Aarray[i]} for \texttt{i=0,1,...,batchSize-1} is a \texttt{n\times{n}} Hermitian matrix, only lower or upper part is meaningful. The input parameter \texttt{uplo} indicates which part of the matrix is used.

If input parameter \texttt{uplo} is \texttt{CUBLAS\_FILL\_MODE\_LOWER}, only lower triangular part of \texttt{A} is processed, and replaced by lower triangular Cholesky factor \texttt{L}. 

\[ A = L^H L \]

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

\[ A = U^H U \]

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 cholesky 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><code>uplo</code></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><code>n</code></td>
<td>host</td>
<td>input</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td><code>Aarray</code></td>
<td>device</td>
<td>in/out</td>
<td>array of pointers to <code>&lt;type&gt;</code> array of dimension <code>lda * n</code> with <code>lda</code> is not less than <code>max(1,n)</code>.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store each matrix <code>Aarray[i]</code>.</td>
</tr>
<tr>
<td><code>infoArray</code></td>
<td>device</td>
<td>output</td>
<td>array of size <code>batchSize</code>. <code>infoArray[i]</code> contains information of factorization of <code>Aarray[i]</code>. If <code>potrfBatched</code> returns <code>CUSOLVER_STATUS_INVALID_VALUE</code>, <code>infoArray[0] = -i</code> (less than zero) means the ( i )-th parameter is wrong (not counting handle). If <code>potrfBatched</code> returns <code>CUSOLVER_STATUS_SUCCESS</code>, <code>infoArray[i] = 0</code> means the Cholesky factorization of ( i )-th matrix is successful, and <code>infoArray[i] = k</code> means the leading submatrix of order ( k ) of ( i )-th matrix is not positive definite.</td>
</tr>
<tr>
<td><code>batchSize</code></td>
<td>host</td>
<td>input</td>
<td>number of pointers in <code>Aarray</code>.</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 ( lda&lt;\max(1,n) ) or ( batchSize&lt;1 )).</td>
</tr>
<tr>
<td>cuSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
5.2.11. cuSolverDn<t>potrsBatched()

This function solves a sequence of linear systems

\[ A[i]X[i] = B[i] \]
where each $A_{array}[i]$ for $i=0,1,...,\text{batchSize}-1$ 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 user has to call potrfBatched first to factorize matrix $A_{array}[i]$. If input parameter uplo is CUBLAS_FILL_MODE_LOWER, $A$ is lower triangular Cholesky factor $L$ correspoding to $A = L*L^H$. If input parameter 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 $\text{ldb}$.

The output parameter 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: infoArray from potrfBatched indicates if the matrix is positive definite. info from 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 uplo is CUBLAS_FILL_MODE_UPPER, upper triangle of $A$ contains cholesky factor $U$ and lower triangle of $A$ is destroyed after potrsBatched.

API of 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>$A_{array}$</td>
<td>device</td>
<td>in/out</td>
<td>array of pointers to &lt;type&gt; array of dimension $\text{lda} * n$ with $\text{lda}$ is not less than max$(1,n)$. $A_{array}[i]$ 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 each matrix $A_{array}[i]$.</td>
</tr>
<tr>
<td>$B_{array}$</td>
<td>device</td>
<td>in/out</td>
<td>array of pointers to &lt;type&gt; array of dimension $\text{ldb} * \text{nrhs}$. $\text{ldb}$ is not less than max$(1,n)$. As an input, $B_{array}[i]$ is right hand side matrix. As an output, $B_{array}[i]$ is the solution matrix.</td>
</tr>
<tr>
<td>$\text{ldb}$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store each matrix $B_{array}[i]$.</td>
</tr>
<tr>
<td>info</td>
<td>device</td>
<td>output</td>
<td>if $\text{info} = 0$, all parameters are correct. if $\text{info} = -i$, 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, nrhs&lt;0, lda&lt;max(1,n), ldb&lt;max(1,n) or batchSize&lt;0).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

5.3. Dense Eigenvalue Solver Reference

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

5.3.1. 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, 
    int lda, 
    float *D, 
    float *E, 
    cuComplex *TAUQ, 
    cuComplex *TAUP, 
    cuComplex *Work, 
    int Lwork, 
    int *devInfo );

cusolverStatus_t
cusolverDnZgebrd(cusolverDnHandle_t handle, 
    int m, 
    int n, 
    cuDoubleComplex *A, 
    int lda, 
    double *D, 
    double *E, 
    cuDoubleComplex *TAUQ, 
    cuDoubleComplex *TAUP, 
    cuDoubleComplex *Work, 
    int Lwork, 
    int *devInfo );
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 \( \text{Lwork} \) is size of the working space, and it is returned by \( \text{gebrd_bufferSize()} \).

If output parameter \( \text{devInfo} = -i \) (less than zero), the \( i \)-\text{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;\text{type}&gt;) 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>( 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 \geq n ), ( E(i) = A(i, i+1) ) for ( i = 1, 2, \ldots, n-1 ); if ( m &lt; n ), ( E(i) = A(i+1, i) ) for ( i = 1, 2, \ldots, m-1 ).</td>
</tr>
<tr>
<td>( \text{TAUQ} )</td>
<td>device</td>
<td>output</td>
<td>(&lt;\text{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>( \text{TAUP} )</td>
<td>device</td>
<td>output</td>
<td>(&lt;\text{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>-----------</td>
<td>----------</td>
<td>--------</td>
<td>------------------------------------------</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**

<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&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>
5.3.2. `cusolverDn<t>orgbr()`

These helper functions calculate the size of work buffers needed.

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

```c
 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,
    int n,
    int k,
    double *A,
    int lda,
    const double *tau,
    double *work,
    int lwork,
    int *devInfo);
```
The C and Z data types are complex valued single and double precision, respectively.

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

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

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 \texttt{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>
<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 side = CUBLAS_SIDE_LEFT, ( m\geq n \Rightarrow m = \min(m,k) ). if side = CUBLAS_SIDE_RIGHT, ( n\geq m \Rightarrow \min(n,k) ).</td>
</tr>
<tr>
<td>k</td>
<td>host</td>
<td>input</td>
<td>if side = CUBLAS_SIDE_LEFT, the number of columns in the original ( m )-by-( k ) matrix reduced by \texttt{gebrd} if side</td>
</tr>
</tbody>
</table>
= CUBLAS\_SIDE\_RIGHT, the number of rows in the original k-by-n matrix reduced by gebrd.

<table>
<thead>
<tr>
<th>A</th>
<th>device</th>
<th>in/out</th>
<th>&lt;type&gt; array of dimension lda * n On entry, the vectors which define the elementary reflectors, as returned by gebrd. On exit, the m-by-n matrix Q or P**T.</th>
</tr>
</thead>
</table>

| lda | host | input | leading dimension of two-dimensional array used to store matrix A. lda >= max(1,m); |

| tau | device | output | <type> array of dimension min(m,k) if side is CUBLAS\_SIDE\_LEFT; of dimension min(n,k) if side is 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 gebrd in its array argument TAUQ or TAUP. |

| work | device | in/out | working space, <type> array of size lwork. |

| lwork | host | input | size of working array work. |

| devInfo | device | output | if info = 0, the ormqr is successful. if info = -i, the i-th parameter is wrong (not counting handle). |

**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 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>
5.3.3. cusolverDn<t>sytrd()

These helper functions calculate the size of work buffers needed.

cusolverStatus_t
cusolverDnSsytrd_bufferSize(  
cusolverDnHandle_t handle,  
cublasFillMode_t uplo,  
int n,  
const float *A,  
int lda,  
const float *d,  
const float *e,  
const float *tau,  
int *lwork);

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

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

cusolverStatus_t
cusolverDnZhetrd_bufferSize(  
cusolverDnHandle_t handle,  
cublasFillMode_t uplo,  
int n,  
const cuDoubleComplex *A,  
int lda,  
const double *d,  
const double *e,  
const cuDoubleComplex *tau,  
int *lwork);
The S and D data types are real valued single and double precision, respectively.

cusolverStatus_t
cusolverDnSsytrd(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    float *A,
    int lda,
    float *d,
    float *e,
    float *tau,
    float *work,
    int lwork,
    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 \( lwork \) is size of the working space, and it is returned by \( \text{sytrd_bufferSize()} \).

If output parameter \( \text{devInfo} = -i \) (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.</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 ( lda \times 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 ), 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 ). ( 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,</td>
</tr>
</tbody>
</table>
\[ E(i) = A(i, i+1) \] if \( \text{uplo} = \text{CUBLAS_FILL_MODE_LOWER} \)

\[ E(i) = A(i+1, i) \] if \( \text{uplo} = \text{CUBLAS_FILL_MODE_UPPER} \)

<table>
<thead>
<tr>
<th>tau</th>
<th>device</th>
<th>output</th>
<th>&lt;type&gt; array of dimension ((n-1)). The scalar factors of the elementary reflectors which represent the orthogonal matrix (Q).</th>
</tr>
</thead>
<tbody>
<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 (\text{sytrd_bufferSize}).</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 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 (\text{lda}&lt;\max(1,n)), or (\text{uplo} ) is not (\text{CUBLAS_FILL_MODE_LOWER}) or (\text{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>
5.3.4. **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,
    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.

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

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

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

\[
C = \begin{cases} 
    \text{op}(Q)^*C & \text{if side} == \text{CUBLAS\_SIDE\_LEFT} \\
    C^*\text{op}(Q) & \text{if side} == \text{CUBLAS\_SIDE\_RIGHT}
\end{cases}
\]

where \( Q \) is a unitary matrix formed by a sequence of elementary reflection vectors from \text{sytrd}.

The operation on \( Q \) is defined by

\[
\text{op}(Q) = \begin{cases} 
    Q & \text{if transa} == \text{CUBLAS\_OP\_N} \\
    Q^T & \text{if transa} == \text{CUBLAS\_OP\_T} \\
    Q^H & \text{if transa} == \text{CUBLAS\_OP\_C}
\end{cases}
\]

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{ormtr\_bufferSize()}. If output parameter \text{devInfo} = -i (less than zero), the \text{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>
<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^T from the Left; side = CUBLAS_SIDE_RIGHT, apply Q or Q^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)<em>C or C</em>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>
</tbody>
</table>
CusolverDN: dense LAPACK Function Reference

<table>
<thead>
<tr>
<th>CUSOLVER_STATUS_INVALID_VALUE</th>
<th>invalid parameters were passed (m, n&lt;0 or wrong lda or ldc).</th>
</tr>
</thead>
<tbody>
<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>

5.3.5. cusolverDn<t>orgtr()

These helper functions calculate the size of work buffers needed.

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,
    float *work,
    int lwork,
    int *devInfo);
```

```c
cusolverStatus_t
cusolverDnDorgtr(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    double *A,
    int lda,
    const double *tau,
    double *work,
    int lwork,
    int *devInfo);
```

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

```c
cusolverStatus_t
cusolverDnCungtr(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    const cuComplex *tau,
    cuComplex *work,
    int lwork,
    int *devInfo);
```

```c
cusolverStatus_t
cusolverDnZungtr(
    cusolverDnHandle_t handle,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    const cuDoubleComplex *tau,
    cuDoubleComplex *work,
    int lwork,
    int *devInfo);
```

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 = CUBLAS_FILL_MODE_UPPER: Upper triangle of A contains elementary reflectors from sytrd.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows (columns) of matrix Q.</td>
</tr>
<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

| 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 wrong lda). |
| CUSOLVER_STATUS_ARCH_MISMATCH    | the device only supports compute capability 2.0 and above. |
| CUSOLVER_STATUS_INTERNAL_ERROR   | an internal operation failed. |
5.3.6. 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,
    int *lwork);

cusolverStatus_t
cusolverDnZgesvd_bufferSize(
    cusolverDnHandle_t handle,
    int m,
    int n,
    int *lwork);
The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnSgesvd (  
cusolverDnHandle_t handle,
signed char jobu,
signed char jobvt,
int m,
int n,
float *A,
int lda,
float *S,
float *U,
int ldu,
float *VT,
int ldvt,
float *work,
int lwork,
float *rwork,
int *devInfo);

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

cusolverStatus_t
cusolverDnCgesvd (  
cusolverDnHandle_t handle,  
signed char jobu,  
signed char jobvt,  
int m,  
int n,  
cuComplex *A,  
int lda,  
float *S,  
cuComplex *U,  
int ldu,  
cuComplex *VT,  
int ldvt,  
cuComplex *work,  
int lwork,  
float *rwork,  
int *devInfo);

cusolverStatus_t
cusolverDnZgesvd (  
cusolverDnHandle_t handle,  
signed char jobu,  
signed char jobvt,  
int m,  
int n,  
cuDoubleComplex *A,  
int lda,  
double *S,  
cuDoubleComplex *U,  
int ldu,  
cuDoubleComplex *VT,  
int ldvt,  
cuDoubleComplex *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 \texttt{work}. The input parameter \texttt{lwork} is size of the working space, and it is returned by \texttt{gesvd_bufferSize()}. If output parameter \texttt{devInfo} = \(-i\) (less than zero), the \( i\)-th parameter is wrong (not counting handle). if \texttt{bdsqr} did not converge, \texttt{devInfo} specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.
The `rwork` is real array of dimension \((\min(m,n)-1)\). If `devInfo>0` and `rwork` is not nil, `rwork` contains the un converged superdiagonal elements of an upper bidiagonal matrix. This is slightly different from LAPACK which puts un converged superdiagonal elements in `work` if type is `real`; in `rwork` if type is `complex`. `rwork` can be a NULL pointer if the user does not want the information from superdiagonal.

Appendix G.1 provides a simple example of `gesvd`.

Remark 1: `gesvd` only supports \(m \geq n\).

Remark 2: the routine returns \(V^H\), not \(V\).

### API of `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 overwritten on the array (A); = 'N': no columns of (U) (no left singular vectors) are computed.</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 overwritten 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;\text{type}&gt;) 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>(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>(U)</td>
<td>device</td>
<td>output</td>
<td>(&lt;\text{type}&gt;) array of dimension (ldu \times 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>
</tbody>
</table>
VT  | device | output | <type> array of dimension ldvt * n with ldvt is not less than max(1,n). VT contains the n×n unitary matrix V**T.

ldvt | host | input | leading dimension of two-dimensional array used to store matrix Vt.

work | device | in/out | working space, <type> array of size lwork.

lwork | host | input | size of work, returned by gesvd_bufferSize.

rwork | device | input | real array of dimension min(m,n)-1. It contains the unconverged superdiagonal elements of an upper bidiagonal matrix if devInfo > 0.

devInfo | device | output | if devInfo = 0, the operation is successful, if devInfo = -i, the i-th parameter is wrong (not counting handle). if devInfo > 0, devInfo indicates how many superdiagonals of an intermediate bidiagonal form did not converge to zero.

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) or ldu<max(1,m) or ldvt<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.
5.3.7. `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,
    gesvdjInfo_t params);

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,
    const double *V,
    int ldv,
    int *lwork,
    gesvdjInfo_t params);

cusolverStatus_t
cusolverDnCgesvdj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    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);

cusolverStatus_t
cusolverDnZgesvdj_bufferSize(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    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);
```
The S and D data types are real valued single and double precision, respectively.

```c
 cusolverStatus_t
cusolverDnSgesvdj(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    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);

cusolverStatus_t
cusolverDnDgesvdj(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int econ,
    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);
```
The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
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);

cusolverStatus_t
cusolverDnZgesvdj(
cusolverDnHandle_t handle,
cusolverEigMode_t jobz,
int econ,
int m,
int n,
cuDoubleComplex *A,
int lda,
double *S,
cuDoubleComplex *U,
int ldu,
cuDoubleComplex *V,
int ldv,
cuDoubleComplex *work,
int lwork,
int *info,
gesvdjInfo_t params);

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 \( A \).

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

$$||E||_F \leq \text{eps} \times ||A||_F$$

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 work. The input parameter lwork is the size of the working space, and it is returned by `gesvdj_bufferSize()`.

If output parameter info = -i (less than zero), the i-th parameter is wrong (not counting handle). If 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 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>econ</td>
<td>host</td>
<td>input</td>
<td>econ = 1 for economy size for $U$ and $V$.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Direction</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</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>
<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 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>
5.3.8. cusolverDn<\textt>gesvdjBatched()

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

```c
 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 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.

cusolverStatus_t
cusolverDnSgesvdjBatched(
  cusolverDnHandle_t handle,
  cusolverEigMode_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);

cusolverStatus_t
cusolverDnDgesvdjBatched(
  cusolverDnHandle_t handle,
  cusolverEigMode_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.

cusolverStatus_t
cusolverDnCgesvdjBatched(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    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,
    int batchSize);

cusolverStatus_t
cusolverDnZgesvdjBatched(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    int m,
    int n,
    cuDoubleComplex *A,
    int lda,
    double *S,
    cuDoubleComplex *U,
    int ldu,
    cuDoubleComplex *V,
    int ldv,
    cuDoubleComplex *work,
    int lwork,
    int *info,
    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,

$$A = (A_0 \ A_1 \ \cdots)$$
Each matrix is column-major with leading dimension \( \text{lda} \), so the formula for random access is \( A_k(i,j) = A[i + \text{lda}\cdot j + \text{lda}\cdot n\cdot k] \).

The parameter \( S \) also contains singular values of each matrix in contiguous way,

\[
S = (S_0 \ S_1 \ \cdots)
\]

The formula for random access of \( S \) is \( S_k(j) = S[j + \min(m,n)\cdot k] \).

Except for tolerance and maximum sweeps, \texttt{gesvdjBatched} can either sort the singular values in descending order (default) or chose as-is (without sorting) by the function \texttt{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.

\texttt{gesvdjBatched} cannot report residual and executed sweeps by function \texttt{cusolverDnXgesvdjGetResidual} and \texttt{cusolverDnXgesvdjGetSweeps}. Any call of the above two returns \texttt{CUSOLVER\_STATUS\_NOT\_SUPPORTED}. The user needs to compute residual explicitly.

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

The output parameter \texttt{info} is an integer array of size \texttt{batchSize}. If the function returns \texttt{CUSOLVER\_STATUS\_INVALID\_VALUE}, the first element \texttt{info[0]} = \(-i\) (less than zero) indicates \( i\)-th parameter is wrong (not counting handle). Otherwise, if \texttt{info[i]} = \( \min(m,n)+1 \), \texttt{gesvdjBatched} does not converge on \( i\)-th matrix under given tolerance and maximum sweeps.

Appendix G.3 provides a simple example of \texttt{gesvdjBatched}.

API of \texttt{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: ( \text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR} ): Compute singular values only; ( \text{jobz} = \text{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 ( A_j ). ( m ) is no greater than 32.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix ( A_j ). ( n ) is no greater than 32.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{type}&gt;) array of dimension ( \text{lda}\cdot n\cdot \text{batchSize} ) with ( \text{lda} ) is not less than ( \max(1,n) ). On Exit: the contents of ( A_j ) 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_j ).</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>device output</td>
<td>a real array of dimension min(m,n) * batchSize. It stores the singular values of A_j in descending order or non-sorting order.</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>device output</td>
<td>&lt;type&gt; array of dimension ldu * m * batchSize. U_j contains the left singular vectors of A_j.</td>
<td></td>
</tr>
<tr>
<td>ldu</td>
<td>host input</td>
<td>leading dimension of two-dimensional array used to store matrix U_j. ldu is not less than max(1,m).</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>device output</td>
<td>&lt;type&gt; array of dimension ldv * n * batchSize. V_j contains the right singular vectors of A_j.</td>
<td></td>
</tr>
<tr>
<td>ldv</td>
<td>host input</td>
<td>leading dimension of two-dimensional array used to store matrix V_j. ldv is not less than max(1,n).</td>
<td></td>
</tr>
<tr>
<td>work</td>
<td>device in/out</td>
<td>&lt;type&gt; array of size lwork, working space.</td>
<td></td>
</tr>
<tr>
<td>lwork</td>
<td>host input</td>
<td>size of work, returned by gesvdjBatched_bufferSize.</td>
<td></td>
</tr>
<tr>
<td>info</td>
<td>device 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, gesvdjBatched does not converge on i-th matrix under given tolerance and maximum sweeps.</td>
<td></td>
</tr>
<tr>
<td>params</td>
<td>host in/out</td>
<td>structure filled with parameters of Jacobi algorithm.</td>
<td></td>
</tr>
<tr>
<td>batchSize</td>
<td>host input</td>
<td>number of matrices.</td>
<td></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 ldu<max(1,m) or ldv<max(1,n) or jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODE_VECTOR, or batchSize<0).
- **CUSOLVER_STATUS_INTERNAL_ERROR**: an internal operation failed.
5.3.9. cusolverDn<t>gesvdaStridedBatched()

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 float *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.

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 parameters $S, U$ and $V$.

The output parameter `h_RnrmF` computes Frobenius norm of residual.

$$A - U \ast S \ast V^H$$

if the parameter `rank` is equal $n$. Otherwise, `h_RnrmF` reports

$$||U \ast S \ast V^H|| - ||S||$$

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,

$$A = \begin{pmatrix} A_0 & A_1 & \cdots \end{pmatrix}$$

Each matrix is column-major with leading dimension `lda`, so the formula for random access is $A_k(i,j) = A[i + \text{lda} \ast j + \text{strideA} \ast k]$. Similarly, the formula for random access of $S$ is $S_k(j) = S[j + \text{strideS} \ast k]$, the formula for random access of $U$ is $U_k(i,j) = U[i + \text{ldu} \ast j + \text{strideU} \ast k]$ and the formula for random access of $V$ is $V_k(i,j) = V[i + \text{ldv} \ast j + \text{strideV} \ast k]$.

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] = -1` (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 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>rank</td>
<td>host</td>
<td>input</td>
<td>number of singular values (from largest to smallest).</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>input</td>
<td>number of rows of matrix Aj.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of columns of matrix Aj.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>input</td>
<td>&lt;type&gt; array of dimension strideA * batchSize with lda is not less than max(1,m). Aj is of dimension m * n.</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>strideA</td>
<td>host</td>
<td>input</td>
<td>value of type long long int that gives the address offset between A[i] and A[i+1]. strideA is not less than lda*n.</td>
</tr>
<tr>
<td>S</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension strideS*batchSize. It stores the singular values of Aj in descending order. Sj is of dimension rank * 1</td>
</tr>
<tr>
<td>strideS</td>
<td>host</td>
<td>input</td>
<td>value of type long long int that gives the address offset between S[i] and S[i+1]. strideS is not less than rank.</td>
</tr>
<tr>
<td>U</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension strideU * batchSize. Uj contains the left singular vectors of Aj. Uj is of dimension m * rank.</td>
</tr>
<tr>
<td>ldu</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix Uj. ldu is not less than max(1,m).</td>
</tr>
<tr>
<td>strideU</td>
<td>host</td>
<td>input</td>
<td>value of type long long int that gives the address offset between U[i] and U[i+1]. strideU is not less than ldu*rank.</td>
</tr>
<tr>
<td>V</td>
<td>device</td>
<td>output</td>
<td>&lt;type&gt; array of dimension strideV * batchSize. Vj contains the right singular vectors of Aj. Vj is of dimension n * rank.</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldv</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix ( V_j ) ( ldv ) is not less than ( \max(1, n) ).</td>
</tr>
<tr>
<td>strideV</td>
<td>host</td>
<td>input</td>
<td>value of type long long int that gives the address offset between ( V[i] ) and ( V[i+1] ). ( strideV ) is not less than ( ldv \times \text{rank} ).</td>
</tr>
<tr>
<td>work</td>
<td>device</td>
<td>in/out</td>
<td>(&lt;\text{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 ( \text{gesvdaStridedBatched_bufferSize} ).</td>
</tr>
<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[0] = 0 ), the operation is successful. If ( info[i] = \min(m, n) + 1 ), ( \text{gesvdaStridedBatched} ) dose not converge on ( i )-th matrix.</td>
</tr>
<tr>
<td>h_RnrmF</td>
<td>host</td>
<td>output</td>
<td>(&lt;\text{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>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&lt;0 ) or ( lda&lt;\max(1, m) ) or ( ldu&lt;\max(1, m) ) or ( ldv&lt;\max(1, n) ) or ( strideA&lt;lda*\text{rank} ) or ( strideS&lt;\text{rank} ) or ( strideU&lt;ldu*\text{rank} ) or ( strideV&lt;ldv*\text{rank} ) or ( batchSize&lt;1 ) or ( jobz ) is not ( \text{CUSOLVER_EIG_MODE_NOVECTOR} ) or ( \text{CUSOLVER_EIG_MODE_VECTOR} ).</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
5.3.10. cusolverDn<t>syevd()

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

```c
 cusolverStatus_t
cusolverDnSsyevd_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);

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);
```

```c
 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);
```

```c
 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

\[
A^*V = V^*A
\]
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 $\text{syevd_bufferSize}()$.

If output parameter $\text{devInfo} = -i$ (less than zero), the $i$-th parameter is wrong (not counting handle). If $\text{devInfo} = i$ (greater than zero), $i$ off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

If $jobz = \text{CUSOLVER_EIG_MODE_VECTOR}$, $A$ 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 $\text{syevd}$.

**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>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 $A$ 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>$&lt;\text{type}&gt;$ array of dimension $\text{lda} \times n$ with $\text{lda}$ is not less than $\text{max}(1, n)$. 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_MODELOWER, 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 $\text{devInfo} = 0$, $A$ contains the orthonormal eigenvectors of the matrix $A$. If $jobz = \text{CUSOLVER_EIG_MODE_NOVECTOR}$, 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>$W$</td>
<td>device</td>
<td>output</td>
<td>$&lt;\text{type}&gt;$ array of dimension $n$. The eigenvalue values of $A$, in ascending order, sorted so that $W(i) \leq W(i+1)$.</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>
</tbody>
</table>
Lwork | host | input | size of work, returned by syevd_bufferSize.
--- | --- | --- | ---
devInfo | device | output | 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 i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

**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 jobz is not CUSOLVER_EIG_MODE_NOVECTOR or CUSOLVER_EIG_MODEVECTOR, 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>
5.3.11. cusolverDn<t>syevdx()

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

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);

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);

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,
    double *A,
    int lda,
    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.

cusolverStatus_t
cusolverDnCheevdx(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    cuComplex *A,
    int lda,
    float vl,
    float vu,
    int il,
    int iu,
    int *h_meig,
    float *W,
    cuComplex *work,
    int lwork,
    int *devInfo);

cusolverStatus_t
cusolverDnZheevdx(
    cusolverDnHandle_t handle,
    cusolverEigMode_t jobz,
    cusolverEigRange_t range,
    cublasFillMode_t uplo,
    int n,
    cuDoubleComplex *A,
    int lda,
    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 $A$. 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 \( \text{jobz} = \text{CUSOLVER\_EIG\_MODE\_VECTOR}, \) \( A \) 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 \text{syevdx}.

**API of \text{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: ( \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>range</td>
<td>host</td>
<td>input</td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: ( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} ) : all eigenvalues/eigenvectors will be found, will becomes the classical \text{syevd/heevd} routine; ( \text{range} = \text{CUSOLVER_EIG_RANGE_V} ) : all eigenvalues/eigenvectors in the half-open interval ((vl,vu]) will be found; ( \text{range} = \text{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. ( \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 ( lda \times n ) with ( 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 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.</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>
<tr>
<td>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</td>
</tr>
</tbody>
</table>
interval to be searched for eigenvalues. \( vl > vu \). Not referenced if \( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} \) 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 subtract/add epsilon (machine precision) to the interval bound such as \( (vl=vl-\epsilon, vu=vu+\epsilon) \). 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_{\text{meig}} )</td>
<td>host</td>
<td>output</td>
<td>integer. The total number of eigenvalues found. ( 0 \leq h_{\text{meig}} \leq n ). If ( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} ), ( h_{\text{meig}} = n ), and if ( \text{range} = \text{CUSOLVER_EIG_RANGE_I} ), ( h_{\text{meig}} = iu-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>
<tr>
<td>( \text{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 ( \text{work} ), returned by ( \text{syevdx} _\text{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 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**

<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 lda&lt;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).</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------------------------------------------------------</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>
5.3.12. cusolverDn<t>sygvd()

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

cusolverStatus_t
cusolverDnSsygvd_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);

cusolverStatus_t
cusolverDnDsygvd_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);

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.

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,
  int *devInfo);
The C and Z data types are complex valued single and double precision, respectively.

cusolverStatus_t
cusolverDnChegvd(
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 *devInfo);

cusolverStatus_t
cusolverDnZhegvd(
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 *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

\[
eig(A, B) = \begin{cases} 
A^*V = B^*V^*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_1} \\
A^*B^*V = V^*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_2} \\
B^*A^*V = V^*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_3}
\end{cases}
\]

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:

\[
\begin{align*}
V^*B^*V &= I & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_1}, \text{CUSOLVER\_EIG\_TYPE\_2} \\
V^*\text{inv}(B)^*V &= I & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_3}
\end{align*}
\]

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{sygvd\_bufferSize()}. If output parameter \texttt{devInfo} = \(-i\) (less than zero), the \(i\)-th parameter is wrong (not counting handle). If \texttt{devInfo} = \(i\) (\(i > 0\) and \(i \leq n\)) and \texttt{jobz} = \texttt{CUSOLVER\_EIG\_MODE\_NOVECTOR}, \(i\) off-diagonal elements of an intermediate tridiagonal form did not converge to zero. If \texttt{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 $\text{jobz} = \text{CUSOLVER\_EIG\_MODE\_VECTOR}$, $A$ 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 $\text{sygvd}$.

### API of $\text{sygvd}$

<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>
<tr>
<td>itype</td>
<td>host</td>
<td>input</td>
<td>Specifies the problem type to be solved: itype = CUSOLVER_EIG_TYPE_1: $A^\top \mathbf{x} = \lambda B \mathbf{x}$. [ itype = CUSOLVER_EIG_TYPE_2: A^\top B \mathbf{x} = \lambda \mathbf{x} ]. [ itype = CUSOLVER_EIG_TYPE_3: B^\top A \mathbf{x} = \lambda \mathbf{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>uplo</td>
<td>host</td>
<td>input</td>
<td>specifies which part of $A$ and $B$ are stored. [ \text{uplo} = \text{CUBLAS_FILL_MODE_LOWER}: \text{Lower triangle of } A \text{ and } B \text{ are stored.} ]. [ \text{uplo} = \text{CUBLAS_FILL_MODE_UPPER}: \text{Upper triangle of } A \text{ and } B \text{ 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 $lda \times n$ with $lda$ 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 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.</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>
| $B$ | device | in/out | $<$type$>$ array of dimension $ldb \times n$. If $\text{uplo} = \text{CUBLAS\_FILL\_MODE\_UPPER}$, the leading $n$-by-$n$ upper triangular part of $B$ contains the upper triangular part of the matrix $B$. If $\text{uplo} = \text{CUBLAS\_FILL\_MODE\_LOWER}$, the leading
n-by-n lower triangular part of B contains the lower triangular part of the matrix B. On exit, if devInfo is less than n, B is overwritten by triangular factor U or L from the Cholesky factorization of B.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Host/Device</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldb</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix B. ldb is not less than max(1,n).</td>
</tr>
<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 sygvd 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). if devInfo = i (&gt; 0), devInfo indicates either potrf or syevd is wrong.</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 ldb&lt;max(1,n), or itype is not 1, 2 or 3, or jobz is not 'N' or 'V', 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>
### 5.3.13. cuSolverDN<t>sygvdx()\

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

```c

cusolverStatus_t
cusolverDnSsygvdx_bufferSize(    
cusolverDnHandle_t handle,    
cusolverEigType_t itype,    
cusolverEigMode_t jobz,    
cusolverEigRange_t range,    
cublasFillMode_t uplo,    
int n,    
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.

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);

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.

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

$$
eig(A, B) = \begin{cases} 
A*V = B*V*A & \text{if } itype = \text{CUSOLVER_EIG_TYPE_1} \\
A^*B*V = V*A & \text{if } itype = \text{CUSOLVER_EIG_TYPE_2} \\
B*A^*V = V*A & \text{if } itype = \text{CUSOLVER_EIG_TYPE_3}
\end{cases}
$$

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., \( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} \)) is requested. The eigenvectors are normalized as follows:

\[
\begin{align*}
V^H B V &= I & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE_1}, \text{CUSOLVER_EIG_TYPE_2} \\
V^H \text{inv}(B) V &= I & \text{if } \text{itype} = \text{CUSOLVER_EIG_TYPE_3}
\end{align*}
\]

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{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 \( \text{jobz} = \text{CUSOLVER_EIG_MODE_VECTOR} \), \( A \) 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 \text{sygvdx}.

API of \text{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: ( \text{itype} = \text{CUSOLVER_EIG_TYPE_1} ): ( A^* x = (\lambda) B^* x ). ( \text{itype} = \text{CUSOLVER_EIG_TYPE_2} ): ( A^* B^* x = (\lambda) A^* x ). ( \text{itype} = \text{CUSOLVER_EIG_TYPE_3} ): ( B^* A^* x = (\lambda) A^* 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: ( \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>range</td>
<td>host</td>
<td>input</td>
<td>specifies options to which selection of eigenvalues and optionally eigenvectors that need to be computed: ( \text{range} = \text{CUSOLVER_EIG_RANGE_ALL} ): all eigenvalues/eigenvectors will be found, will becomes the classical \text{syevd/heevd} routine; ( \text{range} = \text{CUSOLVER_EIG_RANGE_V} ): all eigenvalues/eigenvectors in the half-open interval ( (v_l,v_u] ) will be found; ( \text{range} = \text{CUSOLVER_EIG_RANGE_I} ): the ( i )-th through ( i u )-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. ( \text{uplo} = \text{CUBLAS_FILL_MODE_LOWER} ):</td>
</tr>
</tbody>
</table>

www.nvidia.com

 cuSOLVER Library

DU-06709-001_v10.1 | 114
| Lower triangle of $A$ and $B$ are stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of $A$ and $B$ are stored. |

<table>
<thead>
<tr>
<th><strong>n</strong></th>
<th><strong>host</strong></th>
<th><strong>input</strong></th>
<th>number of rows (or columns) of matrix $A$ and $B$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>device</strong></td>
<td><strong>in/out</strong></td>
<td>$&lt;type&gt;$ array of dimension $lda \times n$ 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.</td>
</tr>
<tr>
<td><strong>lda</strong></td>
<td><strong>host</strong></td>
<td><strong>input</strong></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><strong>B</strong></td>
<td><strong>device</strong></td>
<td><strong>in/out</strong></td>
<td>$&lt;type&gt;$ array of dimension $ldb \times n$. 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 $B$.</td>
</tr>
<tr>
<td><strong>ldb</strong></td>
<td><strong>host</strong></td>
<td><strong>input</strong></td>
<td>leading dimension of two-dimensional array used to store matrix $B$. $ldb$ is not less than $\max(1, n)$.</td>
</tr>
<tr>
<td><strong>vl, vu</strong></td>
<td><strong>host</strong></td>
<td><strong>input</strong></td>
<td>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. $vl &gt; vu$. 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 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...</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(il, iu)</td>
<td>host</td>
<td>input 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 \text{ and } iu = 0) if (n = 0). Not referenced if (\text{range} = \text{CUSOLVER_EIG_RANGE_ALL}) or (\text{range} = \text{CUSOLVER_EIG_RANGE_V}).</td>
</tr>
<tr>
<td>(h_{\text{meig}})</td>
<td>host</td>
<td>output integer. The total number of eigenvalues found. (0 \leq h_{\text{meig}} \leq n). If (\text{range} = \text{CUSOLVER_EIG_RANGE_ALL}), (h_{\text{meig}} = n), and if (\text{range} = \text{CUSOLVER_EIG_RANGE_I}), (h_{\text{meig}} = iu-il+1).</td>
</tr>
<tr>
<td>(W)</td>
<td>device</td>
<td>output a real array of dimension (n). The eigenvalue values of (A), sorted so that (W(i) \geq W(i+1)).</td>
</tr>
<tr>
<td>(\text{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 (\text{work}), returned by (\text{sygvdx_bufferSize}).</td>
</tr>
<tr>
<td>(\text{devInfo})</td>
<td>device</td>
<td>output if (\text{devInfo} = 0), the operation is successful. If (\text{devInfo} = -i), the (i)-th parameter is wrong (not counting handle). If (\text{devInfo} = i) (&gt; 0), (\text{devInfo}) indicates either (\text{potrf}) or (\text{syevd}) is wrong.</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, \text{or } \text{lda}&lt;\text{max}(1,n), \text{or } \text{ldb}&lt;\text{max}(1,n), \text{or } \text{itype}) is not \text{CUSOLVER_EIG_TYPE_1} \text{or } \text{CUSOLVER_EIG_TYPE_2} \text{or } \text{CUSOLVER_EIG_TYPE_3} \text{or } \text{jobz}) is not \text{CUSOLVER_EIG_MODE_NOVECTOR} \text{or } \text{CUSOLVER_EIG_MODE_VECTORL}, \text{or } \text{range} is not \text{CUSOLVER_EIG_RANGE_ALL} \text{or } \text{CUSOLVER_EIG_RANGE_V} \text{or } \text{CUSOLVER_EIG_RANGE_I}, \text{or } \text{uplo} is not \text{CUBLAS_FILL_MODE_LOWER} \text{or } \text{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>
5.3.14. `cusolverDn<t>syevj()`

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

```c
cusolverStatus_t
cusolverDnSyevj_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.

```c
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,
    int lwork,
    int *info,
    syevjInfo_t params);

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

```c
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
\[ A^*Q = Q^*A \]

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.

\texttt{syevj} has the same functionality as \texttt{syevd}. The difference is that \texttt{syevd} uses QR algorithm and \texttt{syevj} uses Jacobi method. The parallelism of Jacobi method gives GPU better performance on small and medium size matrices. Moreover the user can configure \texttt{syevj} to perform approximation up to certain accuracy.

How does it work?

\texttt{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

\[ ||E||_F < \text{eps} \times ||A||_F \]

where \text{eps} is given tolerance.

\texttt{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 \texttt{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 \texttt{cusolverDnXsyevjSetMaxSweeps} to set a proper bound. The experiments show 15 sweeps are good enough to converge to machine accuracy. \texttt{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 \texttt{syevj}, the user can query residual by function \texttt{cusolverDnXsyevjGetResidual} and number of executed sweeps by function \texttt{cusolverDnXsyevjGetSweeps}. However the user needs to be aware that residual is the Frobenius norm of \( E \), not accuracy of individual eigenvalue, i.e.

\[ \text{residual} = ||E||_F = ||A - W||_F \]

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

If output parameter \texttt{info} = \(-i\) (less than zero), the \( i\)-th parameter is wrong (not counting handle). If \texttt{info} = \( n+1 \), \texttt{syevj} does not converge under given tolerance and maximum sweeps.
If the user sets an improper tolerance, \texttt{syevj} may not converge. For example, tolerance should not be smaller than machine accuracy.

if \texttt{jobz} = CUSOLVER\_EIG\_MODE\_VECTOR, \texttt{A} contains the orthonormal eigenvectors \textbf{V}.

Appendix F.3 provides a simple example of \texttt{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>handle</td>
<td>host</td>
<td>input</td>
<td>handle to the cuSolverDN library context.</td>
</tr>
<tr>
<td>\texttt{jobz}</td>
<td>host</td>
<td>input</td>
<td>specifies options to either compute eigenvalue only or compute eigen-pair: \texttt{jobz} = CUSOLVER_EIG_MODE_NOVECTOR: Compute eigenvalues only; \texttt{jobz} = CUSOLVER_EIG_MODE_VECTOR: Compute eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>\texttt{uplo}</td>
<td>host</td>
<td>input</td>
<td>specifies which part of \texttt{A} is stored. \texttt{uplo} = CUBLAS_FILL_MODE_LOWER: Lower triangle of \texttt{A} is stored. \texttt{uplo} = CUBLAS_FILL_MODE_UPPER: Upper triangle of \texttt{A} is stored.</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{A}</td>
<td>device</td>
<td>in/out</td>
<td>&lt;\texttt{type}&gt; array of dimension \texttt{lda} \times \texttt{n} with \texttt{lda} is not less than \texttt{max}(1,\texttt{n}). If \texttt{uplo} = CUBLAS_FILL_MODE_UPPER, the leading \texttt{n}-by-\texttt{n} upper triangular part of \texttt{A} contains the upper triangular part of the matrix \texttt{A}. If \texttt{uplo} = CUBLAS_FILL_MODE_LOWER, the leading \texttt{n}-by-\texttt{n} lower triangular part of \texttt{A} contains the lower triangular part of the matrix \texttt{A}. On exit, if \texttt{jobz} = CUSOLVER_EIG_MODE_VECTOR, and \texttt{info} = 0, \texttt{A} contains the orthonormal eigenvectors of the matrix \texttt{A}. If \texttt{jobz} = CUSOLVER_EIG_MODE_NOVECTOR, the contents of \texttt{A} are destroyed.</td>
</tr>
<tr>
<td>\texttt{lda}</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{W}</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension \texttt{n}. The eigenvalue values of \texttt{A}, in ascending order ie, sorted so that \texttt{W(i)} &lt;= \texttt{W(i+1)}.</td>
</tr>
<tr>
<td>\texttt{work}</td>
<td>device</td>
<td>in/out</td>
<td>working space, &lt;\texttt{type}&gt; array of size \texttt{lwork}.</td>
</tr>
<tr>
<td>\texttt{lwork}</td>
<td>host</td>
<td>input</td>
<td>size of \texttt{work}, returned by \texttt{syevj_bufferSize}.</td>
</tr>
<tr>
<td>\texttt{info}</td>
<td>device</td>
<td>output</td>
<td>if \texttt{info} = 0, the operation is successful. if \texttt{info} = -i, the i-th parameter is wrong (not counting handle). if \texttt{info} = \texttt{n} +1, \texttt{syevj} dose not converge under given tolerance and maximum sweeps.</td>
</tr>
<tr>
<td>\texttt{params}</td>
<td>host</td>
<td>in/out</td>
<td>structure filled with parameters of Jacobi algorithm and results of \texttt{syevj}.</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_MODE_VECTOR, 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>
5.3.15. cusolverDn<t>sygvj()

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

cusolverStatus_t
cusolverDnSsygvj_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,  
syevjInfo_t params);

cusolverStatus_t
cusolverDnDsygvj_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,  
syevjInfo_t params);

cusolverStatus_t
cusolverDnChegvj_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,  
syevjInfo_t params);

cusolverStatus_t
cusolverDnZhegvj_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,  
syevjInfo_t params);
The S and D data types are real valued single and double precision, respectively.

```c
cusolverStatus_t
cusolverDnSsygvj(
    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 *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);
```
The `C` and `Z` data types are complex valued single and double precision, respectively.

```c
 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);
```

```c
 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

\[
eig(A, B) = \begin{cases} 
  A*V = B*V*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_1} \\
  A*B*V = V*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_2} \\
  B*A*V = V*A & \text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_3}
\end{cases}
\]

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:

\[
\begin{align*}
  VH*B*V &= I & &\text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_1}, \text{CUSOLVER\_EIG\_TYPE\_2} \\
  VH*inv(B)*V &= I & &\text{if } \text{itype} = \text{CUSOLVER\_EIG\_TYPE\_3}
\end{align*}
\]

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 $B$,

$$B = LL^H$$

transform the problem to standard eigenvalue problem, then calls `syevj`.

For example, the standard eigenvalue problem of type I is

$$M^* Q = Q^* A$$

where matrix $M$ is symmetric

$$M = L^{-1} A L^{-H}$$

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+1`, `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>
<tr>
<td>itype</td>
<td>host</td>
<td>input</td>
<td>Specifies the problem type to be solved: $\text{itype} = \text{CUSOLVER_EIG_TYPE}_1$: $A^* x = (\lambda) B^* x$. $\text{itype} = \text{CUSOLVER_EIG_TYPE}_2$: $A^* B^* x = (\lambda) x$. $\text{itype} = \text{CUSOLVER_EIG_TYPE}_3$: $B^* A^* x = (\lambda) x$.</td>
</tr>
<tr>
<td>jobz</td>
<td>host</td>
<td>input</td>
<td>Specifies options to either compute eigenvalue only or compute eigen-pair: $\text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR}$: 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 $A$ and $B$ are stored. $\text{uplo} = \text{CUBLAS_FILL_MODE_LOWER}$: Lower triangle of $A$ and $B$ are stored. $\text{uplo} = \text{CUBLAS_FILL_MODE_UPPER}$: Upper triangle of $A$ and $B$ are stored.</td>
</tr>
</tbody>
</table>
### n

**host**
**input**

Number of rows (or columns) of matrix A and B.

### A

**device**
**in/out**

A <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. lda is not less than \( \max(1,n) \).

### B

**device**
**in/out**

B <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 `info` 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 \( \max(1,n) \).

### W

**device**
**output**

A real array of dimension n. The eigenvalue values of A, sorted so that \( W(i) \geq W(i+1) \).

### work

**device**
**in/out**

Working space, <type> array of size lwork.

### lwork

**host**
**input**

Size of work, returned by `sygvj_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 = i (\( i > 0 \)), info indicates either B is not positive definite or `syevj` (called by `sygvj`) does not converge.

---

### 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 (ldb&lt;\max(1, n),) or (iotype) is not 1, 2 or 3, 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).</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
5.3.16. cusolverDn<t>syevjBatched()

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

cusolverStatus_t
cusolverDnSyevjBatched_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,
    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
);
The S and D data types are real valued single and double precision, respectively.

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

```c
int batchSize

A_j^* Q_j = Q_j^* A_j
```

where $A_j$ is a real $n \times n$ diagonal matrix. $Q_j$ is an $n \times n$ unitary matrix. The diagonal elements of $A_j$ are the eigenvalues of $A_j$ in either ascending order or non-sorting order.

`syevjBatched` performs `syevj` on each matrix. It requires that all matrices are of the same size $n$ no greater than 32 and are packed in contiguous way,

```
A = (A0 A1 ⋯)
```

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 W1 ⋯)
```

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, Aj 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 Aj is stored. uplo = CUBLAS_FILL_MODE_LOWER: Lower triangle of Aj is stored. uplo = CUBLAS_FILL_MODE_UPPER: Upper triangle of Aj is stored.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>input</td>
<td>number of rows (or columns) of matrix each 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). If uplo = CUBLAS_FILL_MODE_UPPER, the leading n-by-n upper triangular part of Aj contains the upper triangular part of the matrix Aj. If uplo = CUBLAS_FILL_MODE_LOWER, the leading n-by-n lower triangular part of Aj contains the lower triangular part of the matrix Aj. On exit, if jobz = CUSOLVER_EIG_MODE_VECTOR, and info[j] = 0, Aj contains the orthonormal eigenvectors V_j.</td>
</tr>
</tbody>
</table>
cuSOLVERDN: dense LAPACK Function Reference

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Device</th>
<th>Input/Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eigenvectors of the matrix $A_j$. If $\text{jobz} = \text{CUSOLVER_EIG_MODE_NOVECTOR}$, the contents of $A_j$ are destroyed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{lda}$</td>
<td>host</td>
<td>input</td>
<td>leading dimension of two-dimensional array used to store matrix $A_j$.</td>
<td></td>
</tr>
<tr>
<td>$W$</td>
<td>device</td>
<td>output</td>
<td>a real array of dimension $n*\text{batchSize}$. It stores the eigenvalues of $A_j$ in ascending order or non-sorting order.</td>
<td></td>
</tr>
<tr>
<td>$\text{work}$</td>
<td>device</td>
<td>in/out</td>
<td>&lt;type&gt; array of size $1\text{work}$, workspace.</td>
<td></td>
</tr>
<tr>
<td>$\text{lwork}$</td>
<td>host</td>
<td>input</td>
<td>size of $\text{work}$, returned by $\text{syevjBatched_bufferSize}$.</td>
<td></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>
<td></td>
</tr>
<tr>
<td>$\text{params}$</td>
<td>host</td>
<td>in/out</td>
<td>structure filled with parameters of Jacobi algorithm.</td>
<td></td>
</tr>
<tr>
<td>$\text{batchSize}$</td>
<td>host</td>
<td>input</td>
<td>number of matrices.</td>
<td></td>
</tr>
</tbody>
</table>

**Status Returned**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{CUSOLVER_STATUS_SUCCESS}$</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>$\text{CUSOLVER_STATUS_NOT_INITIALIZED}$</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>$\text{CUSOLVER_STATUS_INVALID_VALUE}$</td>
<td>invalid parameters were passed ($n&lt;0$, $n&gt;32$ or $\text{lda}&lt;\text{max}(1,n)$, 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 $\text{CUBLAS_FILL_MODE_UPPER}$), or $\text{batchSize}&lt;0$.</td>
</tr>
<tr>
<td>$\text{CUSOLVER_STATUS_INTERNAL_ERROR}$</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
This chapter describes the API of cuSolverSP, which provides a subset of LAPACK functions for sparse matrices in CSR or CSC format.

6.1. Helper Function Reference

6.1.1. 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.

Output

| handle | the pointer to the handle to the cuSolverSP context. |

Status Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSOLVER_STATUS_SUCCESS</td>
<td>the initialization succeeded.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_NOT_INITIALIZED</td>
<td>the CUDA Runtime initialization failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ARCH_MISMATCH</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
</tbody>
</table>

6.1.2. cusolverSpDestroy()

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>

### 6.1.3. cusolverSpSetStream()

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>

### 6.1.4. cusolverSpXcsrissym()

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 \( \text{issym} \) 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>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**

<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_ALLOC_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed ($m, \text{nnzA} &lt;= 0$), 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>

### 6.2. 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.
6.2.1. 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,
   const float *b,
   float tol,
   int reorder,
   float *x,
   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,
   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,
   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 \( n \times n \) 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 \( n \), and \( \mathbf{x} \) 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{csrmetisnd} 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{csrmetisnd} 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 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.

\textbf{Input}

<table>
<thead>
<tr>
<th>parameter</th>
<th>\text{cusolverSp MemSpace}</th>
<th>\text{*Host MemSpace}</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>host</td>
<td>handle to the \text{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>
</tbody>
</table>
**cuSolverSP: sparse LAPACK Function Reference**

- **nnzA**: number of nonzeros of matrix A.
- **descrA**: 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.
- **csrValA**: <type> array of nnzA (= csrRowPtrA(n) - csrRowPtrA(0)) nonzero elements of matrix A.
- **csrRowPtrA**: integer array of n + 1 elements that contains the start of every row and the end of the last row plus one.
- **csrColIndA**: integer array of nnzA (= csrRowPtrA(n) - csrRowPtrA(0)) column indices of the nonzero elements of matrix A.
- **b**: right hand side vector of size n.
- **tol**: tolerance to decide if singular or not.
- **reorder**: no ordering if reorder=0. Otherwise, symrcm, symamd, or csrmetis is used to reduce zero fill-in.

**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 n, 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 U(j,j)≈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_ALLOCS_FAILED</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INVALID_VALUE</td>
<td>invalid parameters were passed (n,nnzA&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>
### 6.2.2. cuSolverSp<

cusolverStatus_t
cusolverSpCsrlsvqr[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
\[ A^x = b \]

\( A \) is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \). \( b \) 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}. 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,

\[ A = Q^R \]

If \( A \) is singular under given tolerance (\( \max(tol, 0) \)), then some diagonal elements of \( R \) is zero, i.e.

\[ |R_{jj}| < tol \text{ for some } j \]

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 \( R(1,1) \approx 0 \).

\text{cusolver} library provides three reordering schemes, \text{symrcm} \text{symamd}, and \text{csrmetisnd} to reduce zero fill-in which dramatically affects the performance of QR factorization.

The input parameter \text{reorder} can enable \text{symrcm} (\text{symamd} or \text{csrmetisnd}) if \text{reorder} is 1 (2, or 3), otherwise, no reordering is performed.

\textbf{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 \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>csrValA</td>
<td>device</td>
<td>host</td>
<td>(&lt;\text{type}&gt;) array of ( 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>
### 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 if singular or not.

### reorder
- **host**
- **host**
  - no ordering if reorder=0. Otherwise, symrcm, symamd, or csrmetisnd is used to reduce zero fill-in.

### 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 = 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

<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_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>
6.2.3. 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,
    int reorder,
    double *x,
    int *singularity);

cusolverStatus_t
cusolverSpCcsrlsvchol[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrVal,
    const int *csrRowPtr,
    const int *csrColInd,
    const cuComplex *b,
    float tol,
    int reorder,
    cuComplex *x,
    int *singularity);

cusolverStatus_t
cusolverSpZcsrlsvchol[Host](cusolverSpHandle_t handle,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrVal,
    const int *csrRowPtr,
    const int *csrColInd,
    const cuDoubleComplex *b,
    double tol,
    int reorder,
    cuDoubleComplex *x,
    int *singularity);
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 \texttt{CUSPARSE_MATRIX_TYPE_GENERAL} and upper triangular part of \(A\) is ignored (if parameter \texttt{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 \texttt{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 \texttt{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. \texttt{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 \(G(k,k) \leq \text{tol}\). \texttt{singularity} is the minimum of such \(k\).

\texttt{singularity} is base-0. If \(A\) is positive definite and not near singular under tolerance, \texttt{singularity} is -1. If the user wants to know if \(A\) is positive definite or not, \texttt{tol=0} is enough.

\texttt{cusolver} library provides three reordering schemes, \texttt{symrcm} \texttt{symamd}, and \texttt{csrmetisnd} to reduce zero fill-in which dramatically affects the performance of Cholesky factorization. The input parameter \texttt{reorder} can enable \texttt{symrcm, symamd} or \texttt{csrmetisnd}) if \texttt{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 \texttt{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>
</tbody>
</table>
### descrA
- **host**
- **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`.

### csrValA
- **device**
- **host**
  - `<type>` array of `nnz (= csrRowPtrA(m) - csrRowPtrA(0))` nonzero elements of matrix A.

### csrRowPtrA
- **device**
- **host**
  - Integer array of `m + 1` elements that contains the start of every row and the end of the last row plus one.

### 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 singularity.

### reorder
- **host**
- **host**
  - No ordering if `reorder`=0. Otherwise, `symrcm`, `symamd`, or `csrmetisnd` is used to reduce zero fill-in.

### 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 host</td>
<td></td>
<td>Solution vector of size <code>m</code>, <code>x = inv(A)*b</code>.</td>
</tr>
<tr>
<td>singularity</td>
<td>host host</td>
<td></td>
<td>-1 if A is symmetric positive definite.</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 (<code>m</code>, <code>nnz</code>&lt;0, base index is not 0 or 1, <code>reorder</code> 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>
6.2.4. 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,
   int 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,
   int 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,
   int 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,
   int 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

$$x = \text{argmin} \|Ax - b\|$$

$A$ 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 \texttt{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:

$$A^*P^T = (A_1 \ A_2) = (Q_1 \ Q_2) \begin{pmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix}$$
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 \( \text{tol} \) 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 \( c = Q^*b \), the least square problem can be reformed by

\[
\min ||Ax - b|| = \min ||R^*y - c||
\]

or in matrix form

\[
\begin{pmatrix}
R_{11} & R_{12} \\
R_{22}
\end{pmatrix}
\begin{pmatrix}
Y_1 \\
Y_2
\end{pmatrix}
= \begin{pmatrix}
c_1 \\
c_2
\end{pmatrix}
\]

The output parameter \( \text{min}\_\text{norm} \) 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

\[
\min ||y||
\]

subject to \( R_{11}y_1 + R_{12}y_2 = c_1 \)

Or equivalently another least-square problem

\[
\min ||\begin{pmatrix}
R_{11} & R_{12} \\
I
\end{pmatrix}y_2 - \begin{pmatrix}
R_{11} \backslash c_1 \\
O
\end{pmatrix}||
\]

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>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 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 ( \text{CUSPARSE_MATRIX_TYPE_GENERAL} ). Also, the supported index bases are</td>
</tr>
</tbody>
</table>
### csrValA
- **device**
- **host**
  - `<type>` array of `nnz (= csrRowPtrA(m) - csrRowPtrA(0))` nonzero elements of matrix A.

### csrRowPtrA
- **device**
- **host**
  - Integer array of `m + 1` elements that contains the start of every row and the end of the last row plus one.

### 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>
<tbody>
<tr>
<td>rankA</td>
<td>host</td>
<td>host</td>
<td>Numerical rank of A.</td>
</tr>
<tr>
<td>x</td>
<td>device</td>
<td>host</td>
<td>Solution vector of size n, x=pinv(A)*b.</td>
</tr>
<tr>
<td>p</td>
<td>device</td>
<td>host</td>
<td>A vector of size n, which represents the permutation matrix P satisfying A<em>P^T=Q</em>R.</td>
</tr>
<tr>
<td>min_norm</td>
<td>host</td>
<td>host</td>
<td></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,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>
6.2.5. cusolverSp<
>csreigvsi()

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

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,
    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.

```c
void cusolverSpCcsreigvsi(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);
```

```c
void 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 $A\mathbf{x} = \lambda \mathbf{x}$ 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 $\mathbf{x}$ is the approximated eigenvector of size $m$.

The following shift-inverse method corrects eigenpair step-by-step until convergence. It accepts several parameters:

- $\mu_0$ is an initial guess of eigenvalue. The shift-inverse method will converge to the eigenvalue $\mu$ nearest $\mu_0$ if $\mu$ is a singleton. Otherwise, the shift-inverse method may not converge.

- $\mathbf{x}_0$ is an initial eigenvector. If the user has no preference, just chose $\mathbf{x}_0$ randomly. $\mathbf{x}_0$ must be nonzero. It can be non-unit length.

- $\text{tol}$ is the tolerance to decide convergence. If $\text{tol}$ is less than zero, it would be treated as zero.

- $\text{maxite}$ 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 \( x_0 \):
\[
x^{(0)} = x_0 \text{ of unit length}
\]
for \( j = 0 : \text{maxite} \):
\[
\text{solve} \quad (A - \mu_0 \cdot I) x^{(k+1)} \quad \text{normalize} \quad x^{(k+1)} \quad \text{to unit length}
\]
compute approx. eigenvalue \( \mu = x^H \cdot A \cdot x^{(k+1)} - \mu_0 \)
if \( || \)
endfor

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 number 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 \( |\mu_0| \cdot \text{eps} \), where eps is machine zero. Otherwise, shift-inverse may not converge because of small tolerance.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>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 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 ). 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{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) \text{ nonzero elements of matrix } A. &gt;)</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)) \text{ column indices of the nonzero elements of matrix } A. &gt;)</td>
</tr>
<tr>
<td>( \mu_0 )</td>
<td>host</td>
<td>host</td>
<td>initial guess of eigenvalue.</td>
</tr>
<tr>
<td>( x_0 )</td>
<td>device</td>
<td>host</td>
<td>initial guess of eigenvector, a vector of size ( m ).</td>
</tr>
<tr>
<td>maxite</td>
<td>host</td>
<td>host</td>
<td>maximum iterations in shift-inverse method.</td>
</tr>
</tbody>
</table>
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 ( \mu_0 ) 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>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 ( (m, nnz) \leq 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>
6.2.6. cusolverSp<t>csreigs()

This function computes number of algebraic eigenvalues in a given box \( B \) by contour integral

\[
\text{number of algebraic eigenvalues in box } B = \frac{1}{2 \pi i} \int_{C} \frac{P(z)}{P'(z)} \, dz
\]
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 `left_bottom_corner`) and the other is right upper corner (input parameter `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 `csrValA`, `csrRowPtrA`, and `csrColIndA`.

The output parameter `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 `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 $[0, 4^*N^2]$" where $N$ is number of grids. In this case, `csreigs` is useful for any interval inside $[0, 4^*N^2]$.

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><code>cusolverSpMemSpace</code></th>
<th><code>*HostMemSpace</code></th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>host</td>
<td>host</td>
<td>handle to the cuSolverSP library context.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>host</td>
<td>host</td>
<td>number of rows and columns of matrix $A$.</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>host</td>
<td>host</td>
<td>number of nonzeros of matrix $A$.</td>
</tr>
<tr>
<td><code>descrA</code></td>
<td>host</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td><code>csrValA</code></td>
<td>device</td>
<td>host</td>
<td>$&lt;type&gt;$ array of $nnz (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td><code>csrRowPtrA</code></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><code>csrColIndA</code></td>
<td>device</td>
<td>host</td>
<td>integer array of $nnz (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ column indices of the nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td><code>left_bottom_corner</code></td>
<td>host</td>
<td>host</td>
<td>left bottom corner of the box.</td>
</tr>
<tr>
<td><code>right_upper_corner</code></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

<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_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 \leq 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>

### 6.3. Low Level Function Reference

This section describes low level API of cuSolverSP, including symrcm and batched QR.

#### 6.3.1. cusolverSpXcsrsymrcm()

```c
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:

\[
A(p, p) = P^* A^* P^T
\]

\( A \) 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 \text{CUSPARSE_MATRIX_TYPE_GENERAL}. Internally \text{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 \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>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 ( \text{nnzA} ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>hsolver</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

<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 ((n, \text{nnzA} &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>
6.3.2. `cusolverSpXcsrsymmdq()`

```c
 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:

\[
A(p,p) = P^T A P
\]

\( A \) 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 `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 <code>cuSolverSP</code> 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 <code>csrValA</code> and <code>csrColIndA</code>.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
</tbody>
</table>
**6.3.3. cusolverSpXcsrsymamd()**

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 \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \).

The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}. Internally \texttt{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 ( \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 \texttt{CUSPARSE_INDEX_BASE_ZERO} and \texttt{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 ( \text{nnzA} ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>hSolver</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

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{CUSOLVER_STATUS_SUCCESS}</td>
<td>the operation completed successfully.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_NOT_INITIALIZED}</td>
<td>the library was not initialized.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_ALLOC_FAILED}</td>
<td>the resources could not be allocated.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_INVALID_VALUE}</td>
<td>invalid parameters were passed ((n, \text{nnzA} \leq 0)), base index is not 0 or 1.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_ARCH_MISMATCH}</td>
<td>the device only supports compute capability 2.0 and above.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_INTERNAL_ERROR}</td>
<td>an internal operation failed.</td>
</tr>
<tr>
<td>\texttt{CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED}</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>
6.3.4. cusolverSpXcsrmetisnd()

cusolverStatus_t
cusolverSpXcsrmetisndHost(
    cusolverSpHandle_t handle,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const int64_t *options,
    int *p);

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 `options` is the configuration of `metis`. For those who do not have experiences of `metis`, set `options = 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 `csrValA`, `csrRowPtrA`, and `csrColIndA`.

The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`. Internally `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>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix ( A ). It is the size of <code>csrValA</code> and <code>csrColIndA</code>.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix ( A ). The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</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>Parameter</td>
<td>Host MemSpace</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>host</td>
<td>integer array of ( \text{nnzA} ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>options</td>
<td>host</td>
<td>integer array to configure \text{metis}.</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>
</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_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)), base index is not 0 or 1.</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>
6.3.5. cusolverSpXcsrZfd()

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 \( csrValA, csrRowPtrA, \) and \( csrColIndA \). The supported matrix type is \texttt{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:

\[
A(p,:) = P^* A
\]

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>( \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>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{csrValA} )</td>
<td>host</td>
<td>&lt;type&gt; array of ( \text{nnzA} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{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>( \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>*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>( \text{numnz} )</td>
<td>host</td>
<td>number of nonzeros on diagonal of permuted matrix.</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)), 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>
</tbody>
</table>
### 6.3.6. 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,
   const int *p,
   const int *q,
   int *map,
   void *pBuffer);
```

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

$$B = P A Q^T$$

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

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 permutate matrix $A$.

This function only computes sparsity pattern of $B$. The user can use parameter $map$ to get $csrValB$ as well. The parameter $map$ is an input/output. If the user sets $map=0:1:(nnzA-1)$ before calling $csrperm$, $csrValB=csrValA(map)$.

The supported matrix type is `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 `csrperm_bufferSize()`. The address of `pBuffer` must be a multiple of 128 bytes. If it is not, `CUSOLVER_STATUS_INVALID_VALUE` is returned.

For example, if matrix $A$ is

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

and left permutation vector $p=(0, 2, 1)$, right permutation vector $q=(2, 1, 0)$, then $P^*A^*Q^T$ is

$$P^*A^*Q^T = \begin{pmatrix} 3.0 & 2.0 & 1.0 \\ 9.0 & 8.0 & 7.0 \\ 6.0 & 5.0 & 4.0 \end{pmatrix}$$

Remark 1: only CPU (Host) path is provided.

Remark 2: the user can combine `csrsymrcm` and `csrperm` to get $P^*A^*P^T$ which has less zero fill-in during QR factorization.

**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 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>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix $A$. It is the size of <code>csrValA</code> and <code>csrColIndA</code>.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</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 $B$, <code>map</code> must be set $0:1:(nnzA-1)$.</td>
</tr>
</tbody>
</table>
| pBuffer   | host                | buffer allocated by the user, the size is returned by `csrperm_bufferSize()`.

**Output**
6.3.7. cusolverSpXcsrqrBatched()

The create and destroy methods start and end the lifetime of a csrqrInfo object.

```c
 cusolverStatus_t
cusolverSpCreateCsrqrInfo(csrqrInfo_t *info);
```

```c
 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);
```
Calculate buffer sizes for complex valued data types.

```c
 cusolverStatus_t
cusolverSpCcsrqrBufferInfoBatched(cusolverSpHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int batchSize,
    csrqrInfo_t info,
    size_t *internalDataInBytes,
    size_t *workspaceInBytes);
```

```c
 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.

cusolverStatus_t
cusolverSpScsrqrsBatched(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 nnz,
    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.

```
cusolverStatus_t
 cusolverSpCcsrqrsvBatched(cusolverSpHandle_t handle,
   int m,
   int n,
   int nnzA,
   const cusparseMatDescr_t descrA,
   const cuComplex *csrValA,
   const int *csrRowPtrA,
   const int *csrColIndA,
   const cuComplex *b,
   cuComplex *x,
   int batchSize,
   csrqrInfo_t info,
   void *pBuffer);
```

```
cusolverStatus_t
 cusolverSpZcsrqrsvBatched(cusolverSpHandle_t handle,
   int m,
   int n,
   int nnzA,
   const cusparseMatDescr_t descrA,
   const cuDoubleComplex *csrValA,
   const int *csrRowPtrA,
   const int *csrColIndA,
   const cuDoubleComplex *b,
   cuDoubleComplex *x,
   int batchSize,
   csrqrInfo_t info,
   void *pBuffer);
```

The batched sparse QR factorization is used to solve either a set of least-squares problems

\[ x_j = \arg \min \| A_j^* z - b_j \|, j = 1, 2, \ldots, \text{batchSize} \]

or a set of linear systems

\[ A_j x_j = b_j, j = 1, 2, \ldots, \text{batchSize} \]

where each \( A_j \) is a \( m \times n \) sparse matrix that is defined in CSR storage format by the four arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \) and \( \text{csrColIndA} \).

The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}. If \( A \) is symmetric and only lower/upper part is prvided, 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 \( j = 1, 2, \ldots, \text{batchSize} \).

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 one after another. In other words, \( \text{csrValA}[k*\text{nnzA} : (k+1)*\text{nnzA}] \) is the value of \( A_k \).

The batched QR uses opaque data structure \texttt{csrqrInfo} to keep intermediate data, for example, matrix \( Q \) and matrix \( R \) of QR factorization. The user needs to create \texttt{csrqrInfo} first by \texttt{cusolverSpCreateCsrqrInfo} before any function in batched QR operation.
The `csrqrInfo` would not release internal data until `cusolverSpDestroyCsrqrInfo` is called.

There are three routines in batched sparse QR, `cusolverSpXcsrqrAnalysisBatched`, `cusolverSp[S|D|C|Z]csrqrBufferInfoBatched` and `cusolverSp[S|D|C|Z]csrqrsvBatched`.

First, `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 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]csrqrsvBatched`. 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 \( A_0, A_1, ..., A_9 \) have the same sparsity pattern, the following code solves 10 linear systems \( A_j x_j = b_j, \ j = 0,2,..., 9 \) 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
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);

// 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**
parameter | cuSolverSp MemSpace | description
--- | --- | ---
handle | host | handle to the cuSolverSP library context.
m | host | number of rows of each matrix $A_j$.
$n$ | host | number of columns of each matrix $A_j$.
nnzA | host | number of nonzeros of each matrix $A_j$. It is the size $\text{csrColIndA}$.
descrA | host | the descriptor of each matrix $A_j$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.
csrValA | device | $<\text{type}>$ array of $\text{nnzA} \times \text{batchSize}$ nonzero elements of matrices $A_0, A_1, \ldots$. All matrices are aggregated one after another.
csrRowPtrA | device | integer array of $m+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA | device | integer array of $\text{nnzA}$ column indices of the nonzero elements of each matrix $A_j$.
b | device | $<\text{type}>$ array of $m \times \text{batchSize}$ of right-hand-side vectors $b_0, b_1, \ldots$. All vectors are aggregated one after another.
batchSize | host | number of systems to be solved.
info | host | opaque structure for QR factorization.
pBuffer | device | buffer allocated by the user, the size is returned by $\text{cusolverSpXcsrqrBufferInfoBatched()}$.

Output

| parameter | cuSolverSp MemSpace | description
--- | --- | ---
x | device | $<\text{type}>$ array of $m \times \text{batchSize}$ of solution vectors $x_0, x_1, \ldots$. All vectors are aggregated one after another.
internalDataInBytes | host | number of bytes of the internal data.
workspaceInBytes | host | number of bytes of the buffer in numerical factorization.

Status Returned

| Status Code | Description |
--- | --- |
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 ($m, n, \text{nnzA} < 0$), base index is not 0 or 1. |
<table>
<thead>
<tr>
<th>cuSOLVER Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
Chapter 7.
CUSOLVERRF: REFACTORIZATION REFERENCE

This chapter describes API of cuSolverRF, a library for fast refactorization.

7.1. cusolverRfAccessBundledFactors()

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 $Mi$ and $Mx$. 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>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</td>
</tr>
</tbody>
</table>
row and by column within each row. The array size is $nnzM$.

| $Mx$ | device | output | 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 $nnzM$. |

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_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
</tbody>
</table>

7.2. cusolverRfAnalyze()

cusolverStatus_t

cusolverRfAnalyze(cusolverRfHandle_t handle);

This routine performs the appropriate analysis of parallelism available in the LU refactorization depending upon the algorithm chosen by the user.

$$A = L^*U$$

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

$$A_i x_i = f_i$$

<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

<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_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>an allocation of memory failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>
7.3. cusolverRfSetupDevice()

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

$$ A_1x_1 = f_1 $$

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.

This routine needs to be called only once for a single linear system

$$ A_1x_1 = f_1 $$

<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>
</tbody>
</table>
| csrRowPtrA    | device   | input  | 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 $n+1$. |
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrColIndA</td>
<td>device</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>csrValA</td>
<td>device</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>nnzL</td>
<td>host</td>
<td>the number of non-zero elements of matrix ( L ).</td>
</tr>
<tr>
<td>csrRowPtrL</td>
<td>device</td>
<td>the array of offsets corresponding to the start of each row in the arrays ( \text{csrColIndL} ) and ( \text{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>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 ( \text{nnzL} ).</td>
</tr>
<tr>
<td>csrValL</td>
<td>device</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 ( \text{nnzL} ).</td>
</tr>
<tr>
<td>nnzU</td>
<td>host</td>
<td>the number of non-zero elements of matrix ( U ).</td>
</tr>
<tr>
<td>csrRowPtrU</td>
<td>device</td>
<td>the array of offsets corresponding to the start of each row in the arrays ( \text{csrColIndU} ) and ( \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>csrColIndU</td>
<td>device</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 ( \text{nnzU} ).</td>
</tr>
<tr>
<td>csrValU</td>
<td>device</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 ( \text{nnzU} ).</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>the left permutation (often associated with pivoting). The array size in ( n ).</td>
</tr>
<tr>
<td>Q</td>
<td>device</td>
<td>the right permutation (often associated with reordering). The array size in ( n ).</td>
</tr>
<tr>
<td>handle</td>
<td>host</td>
<td>the handle to the GLU library.</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>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>

7.4. cusolverRfSetupHost()

cusolverStatus_t
cusolverRfSetupHost(/* Input (in the host memory) */
in 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

$$A_1x_1 = f_1$$

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.

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>
</table>

www.nvidia.com

cuSOLVER Library
<table>
<thead>
<tr>
<th>Variable</th>
<th>Host Type</th>
<th>Input Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix $A$.</td>
</tr>
<tr>
<td><code>nnzA</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix $A$.</td>
</tr>
<tr>
<td><code>h_csrRowPtrA</code></td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays <code>h_csrColIndA</code> and <code>h_csrValA</code>. 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><code>h_csrColIndA</code></td>
<td>host</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><code>h_csrValA</code></td>
<td>host</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><code>nnzL</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix $L$.</td>
</tr>
<tr>
<td><code>h_csrRowPtrL</code></td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays <code>h_csrColIndL</code> and <code>h_csrValL</code>. 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><code>h_csrColIndL</code></td>
<td>host</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><code>h_csrValL</code></td>
<td>host</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><code>nnzU</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix $U$.</td>
</tr>
<tr>
<td><code>h_csrRowPtrU</code></td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays <code>h_csrColIndU</code> and <code>h_csrValU</code>. 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><code>h_csrColIndU</code></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</td>
</tr>
</tbody>
</table>
row and by column within each row. The array size is $nnz_U$.

- **h_csrValU**
  - **host**
  - **input**
  - 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 $nnz_U$.

- **h_P**
  - **host**
  - **input**
  - The left permutation (often associated with pivoting). The array size is $n$.

- **h_Q**
  - **host**
  - **input**
  - The right permutation (often associated with reordering). The array size in $n$.

- **handle**
  - **host**
  - **output**
  - The handle to the cuSolverRF library.

### 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>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>

### 7.5. 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

<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>An allocation of memory failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>An internal operation failed.</td>
</tr>
</tbody>
</table>
7.6. 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**

<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>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>
7.7. **cusolverRfExtractSplitFactorsHost()**

```c
int 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>
<tr>
<td>h.Ui</td>
<td>host</td>
<td>output</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 h_nnzL.</td>
</tr>
</tbody>
</table>
The array size is $h_{nnzU}$.

### 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>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>

### 7.8. cusolverRfDestroy()

```c
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 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>

### 7.9. cusolverRfGetMatrixFormat()

```c
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>
<tr>
<td>format</td>
<td>host</td>
<td>output</td>
<td>the enumerated matrix format type.</td>
</tr>
</tbody>
</table>
### 7.10. `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>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>

### 7.11. `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
7.12. cusolverRfGetResetValuesFastMode()

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

<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>

7.13. cusolverRfGet_Algs()

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

<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>

7.14. cusolverRfRefactor()

cusolverStatus_t cusolverRfRefactor(cusolverRfHandle_t handle);
This routine performs the LU re-factorization

\[ A = L^* U \]

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

\[ A_i x_i = f_i \]

<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**

<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_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ZERO_PIVOT</td>
<td>a zero pivot was encountered during the computation.</td>
</tr>
</tbody>
</table>

### 7.15. 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>
</table>
### Parameter Table

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</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></td>
<td>the number of rows (and columns) of matrix ( A ).</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>input</td>
<td></td>
<td>the number of non-zero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>device</td>
<td>input</td>
<td></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></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></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>P</td>
<td>device</td>
<td>input</td>
<td></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></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></td>
<td>the handle to the cuSolverRF library.</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>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>
</tbody>
</table>

### 7.16. 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.
### 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>an enumerated mode parameter is wrong.</td>
</tr>
</tbody>
</table>

### 7.17. `cusolverRfSetNumericProperties()`

```c
 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

<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>

### 7.18. `cusolverRfSetResetValuesFastMode()`

```c
 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.
### 7.19. 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>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>an enumerated mode parameter is wrong.</td>
</tr>
</tbody>
</table>

### 7.20. 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 solves a linear system of equations. It may be called after `cusolverRfRefactor()` and `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>XF</td>
<td>host</td>
<td>input</td>
<td>the problem's solution.</td>
</tr>
<tr>
<td>ldxf</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of <code>XP</code>.</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>
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 \text{cusolverRfRefactor()} routine.

The routine can solve linear systems with multiple right-hand-sides (rhs),

\[
AX = (LU)X = L(UX) = LY = F \quad \text{where} \quad UX = Y
\]

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 nrhs )).</td>
</tr>
<tr>
<td>ldt</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix 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 nrhs )).</td>
</tr>
<tr>
<td>ldxf</td>
<td>host</td>
<td>input</td>
<td>the leading dimension of dense matrix ( XF ) (( ldxf \geq n )).</td>
</tr>
</tbody>
</table>

**Status Returned**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
7.21. cusolverRfBatchSetupHost()

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,
    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 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

\[
A_j x_j = b_j, \quad j = 1, 2, ..., \text{batchSize}
\]

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 \(A_j\).

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

\[
A_i x_i = f_i
\]

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.

Remark 1: the matrices \(A, L\) and \(U\) must be CSR format and base-0.

Remark 2: to get best performance, `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 `batchSize`. In practice, `batchSize` of 32 -
128 is often enough to obtain good performance, but in some cases larger `batchSize` might be beneficial.

This routine needs to be called only once for a single linear system

\[ A_i x_i = f_i \]

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>MemSpace</strong></th>
<th><strong>In/Out</strong></th>
<th><strong>Meaning</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>batchSize</code></td>
<td>host</td>
<td>input</td>
<td>the number of matrices in the batched mode.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>host</td>
<td>input</td>
<td>the number of rows (and columns) of matrix A.</td>
</tr>
<tr>
<td><code>nnzA</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix A.</td>
</tr>
<tr>
<td><code>h_csrRowPtrA</code></td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays <code>h_csrColIndA</code> and <code>h_csrValA</code>. 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 <code>n +1</code>.</td>
</tr>
<tr>
<td><code>h_csrColIndA</code></td>
<td>host</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 <code>nnzA</code>.</td>
</tr>
<tr>
<td><code>h_csrValA_array</code></td>
<td>host</td>
<td>input</td>
<td>array of pointers of size <code>batchSize</code>, each pointer points to the array of values corresponding to the non-zero elements in the matrix.</td>
</tr>
<tr>
<td><code>nnzL</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix L.</td>
</tr>
<tr>
<td><code>h_csrRowPtrL</code></td>
<td>host</td>
<td>input</td>
<td>the array of offsets corresponding to the start of each row in the arrays <code>h_csrColIndL</code> and <code>h_csrValL</code>. 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 <code>n+1</code>.</td>
</tr>
<tr>
<td><code>h_csrColIndL</code></td>
<td>host</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 <code>nnzL</code>.</td>
</tr>
<tr>
<td><code>h_csrValL</code></td>
<td>host</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 <code>nnzL</code>.</td>
</tr>
<tr>
<td><code>nnzU</code></td>
<td>host</td>
<td>input</td>
<td>the number of non-zero elements of matrix U.</td>
</tr>
</tbody>
</table>
h_csrRowPtrU  | host  | input  | 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.

h_csrColIndU  | host  | input  | 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.

h_csrValU    | host  | input  | 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.

h_P          | host  | input  | the left permutation (often associated with pivoting). The array size in n.

h_Q          | host  | input  | the right permutation (often associated with reordering). The array size in n.

handle       | host  | output | the handle to the cuSolverRF library.

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>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>

7.22. cusolverRfBatchAnalyze()

cusolverStatus_t cusolverRfBatchAnalyze(cusolverRfHandle_t handle);

This routine performs the appropriate analysis of parallelism available in the batched LU re-factorization.

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

\[ A_{i,j}x_j = b_{j,} j = 1, 2, ..., batchSize \]
### handle host in/out
the handle to the cuSolverRF library.

#### 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_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_ALLOC_FAILED</td>
<td>an allocation of memory failed.</td>
</tr>
<tr>
<td>CUSOLVER_STATUS_INTERNAL_ERROR</td>
<td>an internal operation failed.</td>
</tr>
</tbody>
</table>

#### 7.23. cusolverRfBatchResetValues()

cusolverStatus_t
cusolverRfBatchResetValues(/* Input (in the device memory) */
int batchSize,
int n,
int nnzA,
int* csrRowPtrA,
int* csrColIndA,
double* csrValA_array[],
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 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

\[ A_j x_j = b_j, j = 1, 2, \ldots, \text{batchSize} \]

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</td>
</tr>
</tbody>
</table>
the number of non-zero elements in the matrix. The array size is $n+1$.

csrColIndA | device | input
|---|---|---|
| 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 $nnzA$.

csrValA_array | device | input
| array of pointers of size batchSize, each pointer points to the array of values corresponding to the non-zero elements in the matrix.

P | device | input
| the left permutation (often associated with pivoting). The array size in $n$.

Q | device | input
| the right permutation (often associated with reordering). The array size in $n$.

handle | host | output
| the handle to the cuSolverRF library.

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>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>
</tbody>
</table>

7.24. cusolverRfBatchRefactor()

cusolverStatus_t cusolverRfBatchRefactor(cusolverRfHandle_t handle);

This routine performs the LU re-factorization

$$M_j = P^*A_j^*Q^T = L_j^*U_j$$

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

<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_EXECUTION_FAILED</td>
<td>a kernel failed to launch on the GPU.</td>
</tr>
</tbody>
</table>
7.25. 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 $M_j Q^* x_j = P^* b_j$
where $M_j = P^* A_j^* Q^T$. Then do refactorization $M_j = L_j U_j$ by
cusolverRfBatch_Refactor(). Further cusolverRfBatch_Solve() takes over the
remaining steps, including:

\[ z_j = P^* b_j \]
\[ M_j y_j = z_j \]
\[ x_j = Q^* y_j \]

The input parameter XF_array is an array of pointers on device memory. 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 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</td>
</tr>
</tbody>
</table>
that contains the right-hand-sides \( F \) and solutions \( X \) (of size \( ldxf \times nrhs \)).

| ldxf | host | input | the leading dimension of dense matrix \( XF \) (\( ldxf \geq n \)). |

**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>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>

### 7.26. `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 \( M_j = P_j A_j Q_j^T = L_j U_j \) 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 \( position(j) \) is -1, the LU refactorization of matrix \( A_j \) is successful. If \( position(j) \) is \( k \geq 0 \), matrix \( A_j \) is not LU factorizable and its matrix \( U_j(j,j) \) 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>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_ZERO_PIVOT</td>
<td>a zero pivot was encountered during the computation.</td>
</tr>
</tbody>
</table>
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.

```c
/* 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.

```c
// 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("cusolverRfReRefactor 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("cusolverRfSolve time = %f (s)\n", t2-t1);
}
```

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
    //after another on the host, then Ax_array[j] = &Ax_batch[nnzA*j];
    //for j=0,...,batchSize-1.
    double *Ax_batch=NULL;
    int *d_Ap=NULL;
    int *d_Ai=NULL;
    //array of pointers to the values of each matrix in the batch (of size
    //batchSize) on the device
    double **d_Ax_array=NULL;
    //For example, if d_Ax_batch is the array (of size batchSize*nnzA)
    containing
    //the values of each matrix in the batch written contiguously one matrix
    //after another on the device, then d_Ax_array[j] = &d_Ax_batch[nnzA*j];
    //for j=0,...,batchSize-1.
    double *d_Ax_batch=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;
```
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,...,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.

```c
//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,...,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;
```
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 perturbations 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
    // 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

    /*
     * | 1 | 2 |
     * | 3 |
     * | 0.1 0.1 0.1 4 |
     * CSR of A is based-1
     * b = [1 1 1 1]
     */
```
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 = cudaMemcpy(d_csrValA, csrValABatch, sizeof(double) * nnzA * batchSize, cudaMemcpyHostToDevice);
.cudaStat2 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA, cudaMemcpyHostToDevice);
.cudaStat3 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1), cudaMemcpyHostToDevice);
.cudaStat4 = cudaMemcpy(d_b, bBatch, sizeof(double) * m * batchSize, cudaMemcpyHostToDevice);
assert(cudaStat1 == cudaSuccess);
assert(cudaStat2 == cudaSuccess);
assert(cudaStat3 == cudaSuccess);
assert(cudaStat4 == cudaSuccess);

// step 4: symbolic analysis
.cusolver_status = cusolverSpXcsrqrAnalysisBatched(cusolverH, m, m, nnzA, descrA, d_csrRowPtrA, d_csrColIndA, info);
assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

// step 5: prepare working space
.cusolver_status = cusolverSpDcsrqrBufferInfoBatched(cusolverH, m, m, nnzA, descrA, d_csrValA, 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 = cudaMemcpy(buffer_qr, size_qr);
Get results back

```c
// step 6: numerical factorization
// assume device memory is big enough to compute all matrices.
cusolver_status = cusolverSpDcsrqrsvBatched(  
cusolverH, m, m, nnzA,  
descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,  
d_b, d_x,  
batchSize,  
info,  
buffer_qr);  
assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

// step 7: check residual  
// xBatch = [x0, x1, x2, ...]  
cudaStat1 = cudaMemcpy(xBatch, d_x, sizeof(double)*m*batchSize,  
cudaMemcpyDeviceToHost);  
assert(cudaStat1 == cudaSuccess);

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;
```

### 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]
    */
```
Create the library handle

```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);
```
Set up the data

```c
// step 3: copy Aj and bj to device
    cudaStat1 = cudaMemcpy(d_csrValA, csrValA, sizeof(double) * nnzA * batchSize);
    cudaStat2 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA);
    cudaStat3 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1));
    cudaStat4 = cudaMemcpy(d_b, b, sizeof(double) * m * batchSize);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);
    assert(cudaStat4 == cudaSuccess);

// don't copy csrValABatch and bBatch because device memory may be big enough
    cudaStat1 = cudaMemcpy(d_csrValA, csrValA, sizeof(double) * nnzA, cudaMemcpyHostToDevice);
    cudaStat2 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) *(nnzA, cudaMemcpyHostToDevice);
    cudaStat3 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1), cudaMemcpyHostToDevice);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);

// step 4: symbolic analysis
    cusolver_status = cusolverSpXcsrqrAnalysisBatched(cusolverH, m, m, nnzA,
        descrA, d_csrRowPtrA, d_csrColIndA,
        info);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

// 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,
            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;
        }
        batchSizeMax *= 2; // double batchSizeMax and try it again.
    }
    batchsizeMax = imin(batchSizeMax, batchSize);
    printf("batchsizeMax = %d\n", batchsizeMax);

    batchSizeMax = 2;
```

Perform analysis and call solve

```c
// 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);

    // step 7: solve Aj*xj = bj
    for(int idx = 0; idx < batchSize; idx += batchSizeMax){
        // current batchSize 'cur_batchSize' is the batchSize used in numerical
        // factorization
        const int cur_batchSize = imin(batchSizeMax, batchSize - idx);
        printf("current batchSize = %d\n", cur_batchSize);
        // copy part of Aj and bj to device
        cudaStat1 = cudaMemcpy(d_csrValA, csrValABatch + idx*nnzA,
            sizeof(double) * nnzA * cur_batchSize, cudaMemcpyHostToDevice);
        cudaStat2 = 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,
            descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
            d_b, d_x,
            cur_batchSize, // WARNING: use current batchSize
            info,
            buffer_qr);
        assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

        // copy part of xj back to host
        cudaStat1 = cudaMemcpy(xBatch + idx*m, d_x,
            sizeof(double) * m * cur_batchSize, cudaMemcpyDeviceToHost);
        assert(cudaStat1 == cudaSuccess);
    }
```
Check results

```c
// 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]);
    }
}

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 = Q*R by geqrf.
Step 2: B := Q^T*B by ormqr.
Step 3: solve R*X = B by trsm.
The usual includes and main definition

```c
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include ormqr_example.cpp
 *   nvcc -o -fopenmp a.out ormqr_example.o -L/usr/local/cuda/lib64 -lcudart -lcublas -lcusolver
 *
#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++){
            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;
    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 |
    | 4 5 6 |
    | 2 1 1 |
    *
    x = (1 1 1)'
    * b = (6 15 4)'
    */
}```
Create the library handle and load the data

double A[lda*m] = { 1.0, 4.0, 2.0, 2.0, 5.0, 1.0, 3.0, 6.0, 1.0};
//    double X[ldb*nrhs] = { 1.0, 1.0, 1.0}; // exact solution
double B[ldb*nrhs] = { 6.0, 15.0, 4.0};
double XC[ldb*nrhs]; // solution matrix from GPU

double *d_A = NULL; // linear memory of GPU
double *d_tau = NULL; // linear memory of GPU
double *d_B = NULL;
int *devInfo = NULL; // info in gpu (device copy)
double *d_work = NULL;
int lwork = 0;

printf("A = (matlab base-1)\n");
printMatrix(m, m, A, lda, "A");
printf("=====\n");
printf("B = (matlab base-1)\n");
printMatrix(m, nrhs, B, ldb, "B");
printf("=====\n");

// step 1: create cusolver/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 * m);
cudaStat2 = cudaMalloc ((void**)&d_tau, sizeof(double) * m);
cudaStat3 = cudaMalloc ((void**)&d_B, sizeof(double) * ldb * nrhs);
cudaStat4 = cudaMalloc ((void**)&devInfo, 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_tau, A, sizeof(double) * m,
cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_B, B, sizeof(double) * ldb * nrhs,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
Call the solver

// step 3: query working space of geqrf and ormqr
  cusolver_status = cusolverDnDgeqrf_bufferSize(
      cusolverH,
      m,
      d_A,
      lda,
      &lwork);
  assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

  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 good or not
  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^T*B
  cusolver_status= cusolverDnDormqr(
      cusolverH,
      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);
C.2. orthogonalization

This chapter provides a simple example in the C programming language of how to do orthogonalization by QR factorization.

\[ A = QR \]

A is a 3x2 dense matrix,
The following code uses three steps:

Step 1: \( A = QR \) 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
", 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;
    const int m = 3;
    const int n = 2;
    const int lda = m;
    /*
    *  | 1 2 |
    *  A = | 4 5 |
    *  | 2 1 |
    */
```
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_orqqr = 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,
        n,
        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,
        n,
        d_A,
        lda,
        d_tau,
        d_work,
        lwork,
        devInfo);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
    assert(cudaSuccess == cudaStat1);
```
Check the results

```c
// check if QR is good or not
cudaStat1 = cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);

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;
cublas_status = cublasDnrm2_v2(
    cublasH,
    n, // number of rows of R
    m, // number of columns of R
    &h_minus_one, /* host pointer */
    d_A, // Q**T
    lda,
    d_A, // Q
    lda,
    &h_one, /* host pointer */
    d_R, // I
    n);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

double dR_nrm2 = 0.0;
cublas_status = cublasDnrm2_v2(
    cublasH, n*n, d_R, 1, &dR_nrm2);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);

printf("|I - Q**T*Q| = %E\n", dR_nrm2);
```
free resources

```c
// 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.

\[ A = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 10.0 \end{pmatrix} \]

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.
/*
* How to compile (assume cuda is installed at /usr/local/cuda/)
* nvcc -c -I/usr/local/cuda/include getrf_example.cpp
* g++ -fopenmp -o a.out getrf_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++) {
            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;
    
    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;
    const int ldb = m;
    /*
    | 1 2 3 |
    * A = | 4 5 6 |
    | 7 8 10 |
    *
    * without pivoting: A = L*U
    * | 1 0 0 | | 1 2 3 |
    * | 4 1 0 |, U = | 0 -3 -6 |
    * | 7 2 1 | | 0 0 1 |
    *
    * with pivoting: P*A = L*U
    * | 0 0 1 |
    * P = | 1 0 0 |
    * | 0 1 0 |
    *
    * | 1 0 0 | | 7 8 10 |
    * | 0.1429 1 0 |, U = | 0 0.8571 1.5714 |
    * | 0.5714 0.5 1 | | 0 0 -0.5 |
*/
```c
#include <cusolverDn.h>

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("=====
");

printf("B = (matlab base-1)\n");
printMatrix(m, 1, B, ldb, "B");
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(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);```

LU Examples

/* 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, 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), cudaMemcpypDeviceToHost);
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");
LU Examples

```c
/* step 5: solve A*X = B */
* | 1 |       | -0.3333 |
* B = | 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\n");
printMatrix(m, 1, X, ldb, "X");
printf("====\n\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 \( A_1 \).
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include batchchol_example.cpp
 *   g++ -o a.out batchchol_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++){
            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;
    cudaError_t cudaStat4 = cudaSuccess;

    const cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
    const int batchSize = 2;
    const int nrhs = 1;
    const int m = 3;
    const int lda = m;
    const int ldb = m;

    //
    //  | 1  2  3 |
    //  A0 = | 2  5  5 | = L0 * L0**T
    //  | 3  5 12 |
    //
    //  | 1.0000 0 0 | = L0 * L0**T
    // where L0 = |
    //  | 2.0000 1.0000 0 |
    //  | 3.0000 -1.0000 1.4142 |
    //
    //  | 1  2  3 |
    //  A1 = | 2  4  5 | is not s.p.d., failed at row 2
    //  | 3  5 12 |
    */
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");
printMatrix(m, m, A0, lda, "A0");
printf("=====
");
printf("A1 = (matlab base-1)\n");
printMatrix(m, m, A1, lda, "A1");
printf("=====
");
printf("B0 = (matlab base-1)\n");
printMatrix(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 = cudaMemcpy(Aarray[0], A0, sizeof(double) * lda * m,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

cudaStat2 = cudaMemcpy(Aarray[1], A1, sizeof(double) * lda * m,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat2);
... cudaStat1 = cudaMemcpy(Barray[0], B0, sizeof(double) * m, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(Barray[1], B0, sizeof(double) * m, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

cudaStat1 = cudaMemcpy(d_Aarray, Aarray, sizeof(double*)*batchSize, cudaMemcpyHostToDevice);
cudaStat2 = cudaMemcpy(d_Barray, Barray, sizeof(double*)*batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
cudaDeviceSynchronize();

/* step 3: Cholesky factorization */
status = cusolverDnDpotrfBatched(
    handle,
    uplo,
    m,
    d_Aarray,
    lda,
    d_infoArray,
    batchSize);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(infoArray, d_infoArray, sizeof(int)*batchSize, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(L0, Aarray[0], sizeof(double) * lda * m, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

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 */
status = cusolverDnDpotrsBatched(
    handle,
    uplo,
    m,
    nrhs, /* only support rhs = 1*/
    d_Aarray,
    lda,
    d_Barray,
    ldb,
    d_infoArray,
    batchSize);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(infoArray, d_infoArray, sizeof(int),
cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(X0 , Barray[0], sizeof(double)*m,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
cudaDeviceSynchronize();

printf("info = %d\n", infoArray[0]);
assert( 0 == infoArray[0] );

printf("X0 = (matlab base-1)\n");
printMatrix(m, 1, X0, ldb, "X0");
printf("=====\n");

/* free resources */
if (d_Aarray    ) cudaFree(d_Aarray);
if (d_Barray    ) cudaFree(d_Barray);
if (d_infoArray ) cudaFree(d_infoArray);
if (handle      ) cusolverDnDestroy(handle);
if (stream      ) cudaStreamDestroy(stream);

cudaDeviceReset();

return 0;
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
/*
 * 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
", 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;
    /*       | 3.5 0.5 0 |
    *   A = | 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("=====
");
    cusp...
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);
    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, 
        &lwork);
    assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

// step 4: compute spectrum
    cusolver_status = cusolverDnDsyevd(
        cusolverH, 
        jobz, 
        uplo, 
        m, 
        d_A, 
        lda, 
        d_W, 
        lwork, 
        devInfo);
    cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
check the result

```c
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

\[
A = \begin{bmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2
\end{bmatrix}
\]
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
", 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("=====
");
```

Examples of Dense Eigenvalue Solver

www.nvidia.com

cuSOLVER Library

DU-06709-001_v10.1 | 242
Examples of Dense Eigenvalue Solver

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);

    cudaMemcpy(d_A, A, sizeof(double) * lda * m, cudaMemcpyHostToDevice);

// 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);
    cudaMemcpy(d_A, A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
    cudaStat1 = cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);

// step 4: compute spectrum
    cusolver_status = cusolverDnDsyevd(   
        cusolverH,
        jobz,  
        uplo,  
        m,    
        d_A,  
        lda,  
        d_W,  
        d_work,  
        lwork,  
        devInfo);
    cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(&info_gpu, devInfo, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(W, d_W, sizeof(double)*m, cudaMemcpyDeviceToHost);
    cudaMemcpy(V, d_A, sizeof(double)*lda*m, cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
check the result

```c
    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.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

\[
A = \begin{pmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2
\end{pmatrix}
\]

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
/*
 How to compile (assume cuda is installed at /usr/local/cuda/)
 nvcc -c -I/usr/local/cuda/include sygvd_example.cpp
 g++ -o a.out sygvd_example.o -L/usr/local/cuda/lib64 -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;
    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, 3.0, 5.0, 10.0};
    double lambda[m] = { 0.158660256604, 0.370751508101882, 0.6};
    double V[lda*m]; // eigenvectors
    double W[m]; // eigenvalues

    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");
    printfMatrix(m, m, A, lda, "A");
    printf("====\n");

    printf("B = (matlab base-1)\n");
    printfMatrix(m, m, B, lda, "B");
    printf("====\n");
}
```
Examples of Dense Eigenvalue Solver

call eigenvalue solver

```c
// 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);

// 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
    cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER; // and eigenvectors.
    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 = cudaMalloc((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 = 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,
cudaMemcpyDeviceToDeviceHost);
cudaStat3 = cudaMemcpy(&info_gpu, devInfo, sizeof(int),
cudaMemcpyDeviceToDeviceHost);
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

\[
B = \begin{bmatrix}
10 & 2 & 3 \\
2 & 10 & 5 \\
3 & 5 & 10
\end{bmatrix}
\]
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
", 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;

    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 V[lda*m]; // eigenvectors
    double W[m]; // eigenvalues

    printf("A = (matlab base-1)\n");
    printMatrix(m, m, A, lda, "A");
    printf("====\n");
    printf("B = (matlab base-1)\n");
    printMatrix(m, m, B, lda, "B");
    printf("====\n");
    return 0;
}
```
call eigenvalue solver

```c
// 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);

// 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);
    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);
    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("====\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.5. Standard Symmetric Dense Eigenvalue Solver (via Jacobi method)

This chapter provides a simple example in the C programming language of how to use 
\texttt{syevj} to compute the spectrum of a dense symmetric system by

\[
A x = \lambda x
\]

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
/*
* How to compile (assume cuda is installed at /usr/local/cuda/)
* nvcc -I/usr/local/cuda/include syevj_example.cpp
* g++ -o syevj_example syevj_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++)
        {
            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;
    /*
     *       | 3.5 0.5 0 |
     *   A = | 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

```c
/* numerical results of syevj */

double residual = 0;
int executed_sweeps = 0;

printf("example of syevj \n");
printf("tol = %E, default value is machine zero \n", 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("=====
");

/* 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 */
cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(double) * lda * m);
cudaStat2 = cudaMalloc ((void**)&d_W, sizeof(double) * m);
cudaStat3 = cudaMalloc ((void**)&d_info, sizeof(int));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*m,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
```
call eigenvalue solver

/* step 4: query working space of syevj */
status = cusolverDnDsyevj_bufferSize(
    cusolverH,
    jobz,
    uplo,
    m,
    d_A,
    lda,
    d_W,
    &lwork,
    syevj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);

/* step 5: compute eigen-pair */
status = cusolverDnDsyevj(
    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 == 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);

if ( 0 == info ){
    printf("syevj 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]);
}

printf("V = (matlab base-1)\n");
printMatrix(m, m, V, lda, "V");
printf("====\n");
check the result

```c
/* 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 |A - 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_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;
}
```


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

\[
A = \begin{bmatrix}
3.5 & 0.5 & 0 \\
0.5 & 3.5 & 0 \\
0 & 0 & 2 \\
\end{bmatrix}
\]

and \(B\) is a 3x3 positive definite matrix
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\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;
    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 V[lda*m]; /* eigenvectors */
    double W[m];     /* eigenvalues  */
    double *d_A = NULL; /* device copy of A */
    double *d_B = NULL; /* device copy of B */
    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
const 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;

printf("example of sygvj \n");
printf("tol = %E, default value is machine zero \n", 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("=====
");

printf("B = (matlab base-1)\n");
printMatrix(m, m, B, lda, "B");
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: 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);
```
call eigenvalue solver

/* 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);
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(d_W, W, sizeof(double)*m, cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(d_A, A, sizeof(double)*lda*m,
cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(d_info, &info, sizeof(int), cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
check the result

```c
if ( 0 == info ){  
    printf("sygvj converges \n");
} else if ( 0 > info ){  
    printf("Error: %d-th parameter is wrong \n", -info);
    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

\[
A_0 = \begin{pmatrix}
1 & -1 & 0 \\
-1 & 2 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

\[
A_1 = \begin{pmatrix}
3 & 4 & 0 \\
4 & 7 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

The following code uses \texttt{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`.

```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;
    cudaError_t cudaStat4 = cudaSuccess;
    const int m = 3; // 1<= m <= 32
    const int llda = m;
    const int batchSize = 2;
    /*
     * | 1 -1 0 |
     * | -1 2 0 |
     * | 0 0 0 |
     * A0 = V0 * W0 * V0**T
     * W0 = diag(0, 0.3820, 2.6180)
     * | 3 4 0 |
     * | 4 7 0 |
     * | 0 0 0 |
     * A1 = V1 * W1 * V1**T
     * W1 = diag(0, 0.5279, 9.4721)
     */
```
Examples of Dense Eigenvalue Solver

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_V = NULL; /* m-by-batchSize */
int* d_info = NULL; /* batchSize */
int 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;

/*
   |  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;

/*
   |  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;
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;
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 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

/* step 4: query working space of syevjBatched */
status = cusolverDnDsyevjBatched_bufferSize(cusolverH, jobz, uplo, m, d_A, lda, d_W, &lwork, syevj_params, batchSize);
assert(CUSOLVER_STATUS_SUCCESS == status);

cudaStat1 = cudaMalloc((void**)&d_work, sizeof(double)*lwork);
assert(cudaSuccess == cudaStat1);
```

/* whole code is omitted */
call eigenvalue solver

```c
/* step 5: compute spectrum of A0 and A1 */
status = cusolverDnDsyevjBatched(
    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);

    cudaStat1 = cudaMemcpy(V    , d_A   , sizeof(double) * lda * m * batchSize,
    cudaMemcpyDeviceToHost);
    cudaStat2 = cudaMemcpy(W    , d_W   , sizeof(double) * m * batchSize      ,
    cudaMemcpyDeviceToHost);
    cudaStat3 = 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 \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 */
    /* if info[i] is not zero, Jacobi method does not converge at i-th matrix. */
        printf("WARNING: matrix %d, info = %d : syevj does not converge \n",
            i, info[i] );
        }
    }

/* Step 6: show eigenvalues and eigenvectors */
double *W0 = W;
double *W1 = W + m;
print("==== \n");
for(int i = 0 ; i < m ; i++)
{
    printf("W0[%d] = %f\n", i, W0[i]);
}
print("==== \n");
for(int i = 0 ; i < m ; i++)
{
    printf("W1[%d] = %f\n", i, W1[i]);
}
print("==== \n");

double *V0 = V;
double *V1 = V + lda*m;
print("V0 = (matlab base-1)\n");
printMatrix(m, m, V0, lda, "V0");
print("V1 = (matlab base-1)\n");
printMatrix(m, m, V1, lda, "V1");
```

cannot query residual and executed sweeps.

```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 \( A - U \Sigma V^T \)
Examples of Singular Value Decomposition

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;
    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_work = NULL;
    double *d_rwork = 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;

    /*
    * How to compile (assume cuda is installed at /usr/local/cuda/)
    * nvcc -c -I/usr/local/cuda/include svd_example.cpp
    * g++ -o a.out svd_example.o -L/usr/local/cuda/lib64 -lcudart -lcublas -lcusolver
    *
    */
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <assert.h>
    #include <cuda_runtime.h>
    #include <cublas_v2.h>
    #include <cusolverDn.h>
... printf("A = (matlab base-1)\n");
printf("=====\n");

// 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_work , sizeof(double)*lwork);
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,
    d_U,
    lda, // ldu
d_VT,
    lda, // ldvt,
d_work,
lwork,
d_rwork,
d_work,
d_workspace,
    devInfo);
cudaStat1 = cudaDeviceSynchronize();
assert (CUSOLVER_STATUS_SUCCESS == cusolver_status);
assert(cudaSuccess == cudaStat1);
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("====\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 = cublasDdsgmm(
cublasH,
CUBLAS_SIDE_LEFT,
n, n, d_VT, lda,
d_S, 1, d_W, lda);
assert(CUBLAS_STATUS_SUCCESS == cublas_status);
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 \texttt{gesvdj}.

\[
A = U \Sigma V^H
\]

$A$ is a 3x2 dense matrix,
/* How to compile (assume cuda is installed at /usr/local/cuda/)
   * nvcc -c -I/usr/local/cuda/include gesvdj_example.cpp
   * g++ -o gesvdj_example gesvdj_example.o -L/usr/local/cuda/lib64 -lcudart -lcusolver
   */
#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 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) = %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;
    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 */
Examples of Singular Value Decomposition

```c
/* configuration of gesvdj */
const double tol = 1.e-7;
const int max_sweeps = 15;
const cusolverEigMode_t jobz = CUSOLVER_EIG_MODE_VECTOR; // compute eigenvectors.
const int econ = 0; /* econ = 1 for economy size */

/* 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("=====
");

/* 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(gesvdj_params,
max_sweeps);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 3: 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));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
assert(cudaSuccess == cudaStat5);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(double)*lda*n,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
```
Examples of Singular Value Decomposition

/* step 4: query workspace of SVD */
status = cusolverDnDgesvdj_bufferSize(
cusolverH,
jobz, /* CUSOLVER_EIG_MODE_NOVECTOR: compute singular values only */
/* CUSOLVER_EIG_MODE_VECTOR: compute singular value and singular
vectors */
econ, /* econ = 1 for economy size */
m, /* number of rows of A, 0 <= m */
n, /* number of columns of A, 0 <= n */
d_A, /* m-by-n */
da, /* leading dimension of A */
d_S, /* min(m,n) */
/* the singular values in descending order */
d_U, /* m-by-m if econ = 0 */
/* m-by-min(m,n) if econ = 1 */
da, /* leading dimension of U, ldu >= max(1,m) */
d_V, /* n-by-n if econ = 0 */
/* n-by-min(m,n) if econ = 1 */
da, /* leading dimension of V, ldv >= max(1,n) */
lwork,
gesvdj_params);
assert(CUSOLVER_STATUS_SUCCESS == status);

/* step 5: compute SVD */
status = cusolverDnDgesvdj(
cusolverH,
jobz, /* CUSOLVER_EIG_MODE_NOVECTOR: compute singular values only */
/* CUSOLVER_EIG_MODE_VECTOR: compute singular value and singular
vectors */
econ, /* econ = 1 for economy size */
m, /* number of rows of A, 0 <= m */
n, /* number of columns of A, 0 <= n */
d_A, /* m-by-n */
da, /* leading dimension of A */
d_S, /* min(m,n) */
/* the singular values in descending order */
d_U, /* m-by-m if econ = 0 */
/* m-by-min(m,n) if econ = 1 */
da, /* leading dimension of U, ldu >= max(1,m) */
d_V, /* n-by-n if econ = 0 */
/* n-by-min(m,n) if econ = 1 */
da, /* leading dimension of V, ldv >= max(1,n) */
lwork,
lwork,
d_info,
gesvdj_params);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSOLVER_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(U, d_U, sizeof(double)*lda*m,
cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(V, d_V, sizeof(double)*lda*n,
cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S, d_S, sizeof(double)*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) cusolver_Destroy(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 \texttt{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 \texttt{gesvdjBatched} to compute singular values and singular vectors.
The user can disable/enable sorting by the function `cusolverDnXgesvdjSetSortEig`.

```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) = %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) */

    A0 = | 1 -1 |
    | 0 0 |

    A0 = U0 * S0 * V0**T
    S0 = diag(2.6180, 0.382)

    A1 = | 3 4 |
    | 0 0 |

    A1 = U1 * S1 * V1**T
    S1 = diag(9.4721, 0.5279)

    /*
    *        |  1  -1  |
    *   A0 = | -1   2  |
    *        |  0   0  |
    */
```
setup matrices $A_0$ and $A_1$

```c
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] */
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 tol = 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;

double *A0 = A;
double *A1 = A + lda*n; /* Aj is m-by-n */

/*
 | 1  -1 |
| -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 |
| 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 \
");
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);

printf("A0 = (matlab base-1)\n");
printf("A1 = (matlab base-1)\n");
```

www.nvidia.com

cuSOLVER Library
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(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 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
call batched singular value solver

/* 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,
    d_work,
    lwork,
    d_info,
    gesvdj_params,
    batchSize
);
cudaStat1 = cudaMemcpy(U    , d_U   , sizeof(double)*ldu*m*batchSize,
cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(V    , d_V   , sizeof(double)*ldv*n*batchSize,
cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(S    , d_S   , sizeof(double)*minmn*batchSize,
cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(&info, d_info, sizeof(int) * batchSize       ,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
assert(cudaSuccess == cudaStat4);
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 */
        /* if info[i] is not zero, Jacobi method does not converge at i-th matrix. */
        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");
printMatrix(n, n, V0, ldv, "V0");
printf("V1 = (matlab base-1)\n");
printMatrix(n, n, V1, ldv, "V1");
```

G.4. SVD approximation

This chapter provides a simple example in the C programming language of how to approximate singular value decomposition by `gesvdaStridedBatched`.

\[ A = U \Sigma V^H \]

\( A_0 \) and \( A_1 \) are a 3x2 dense matrices,

\[
A_0 = \begin{pmatrix}
1.0 & 2.0 \\
4.0 & 5.0 \\
2.0 & 1.0
\end{pmatrix}
\]

\[
A_1 = \begin{pmatrix}
10.0 & 9.0 \\
8.0 & 7.0 \\
6.0 & 5.0
\end{pmatrix}
\]
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 * nvcc -c -I/usr/local/cuda/include gesvda_example.cpp
 * g++ -o gesvda_example gesvda_example.o -L/usr/local/cuda/lib64 -lcudart -lcusolver
 */

#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  |, A1 = |  8 7 |
    *        | 2 1  |       |  6 5 |
    */
    float A[2*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[2*strideU*batchSize]; /* left singular vectors */
    float V[2*strideV*batchSize]; /* right singular vectors */
    float S[2*strideS*batchSize]; /* numerical singular value */
    /* exact singular values */
    float S_exact[2*strideS*batchSize] = {7.065283497082729, 1.040081297712078, 18.839649186929730, 0.260035600289472};
Examples of Singular Value Decomposition

```c
float *d_A = NULL; /* device copy of A */
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 = cusolverDnDggesvdaStridedBatched(
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);
assert(cudaSuccess == cudaStat1);
cudaStat2 = cudaMemcpy(V, d_V, sizeof(float)*strideV*batchSize,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat2);
cudaStat3 = cudaMemcpy(S, d_S, sizeof(float)*strideS*batchSize,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat3);
cudaStat4 = cudaMemcpy(info, d_info, sizeof(int)*batchSize,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat4);
cudaStat5 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat5);

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");
printMatrix(rank, 1, S, n, "S0");
printf("====\n");

printMatrix(m, rank, U, ldu, "U0");
printf("====\n");

printMatrix(n, rank, V, ldv, "V0");
printf("====\n");

float ds_sup = 0;
for(int j = 0; j < n; j++){
    float err = fabs( S[j] - S_exact[j] );
    ds_sup = (ds_sup > err)? ds_sup : err;
}
printf("|S0 - S0_exact|_sup = %E \n", ds_sup);

printf("residual |A0 - U0*S0*V0**H|_F = %E \n", RnrmF[0] );

printMatrix(rank, 1, S + strideS, n, "S1");
printf("====\n");

printMatrix(m, rank, U + strideU, ldu, "U1");
printf("====\n");

printMatrix(n, rank, V + strideV, ldv, "V1");
printf("====\n");

ds_sup = 0;
for(int j = 0; j < n; j++){
    float err = fabs( S[strideS + j] - S_exact[ strideS + j ] );
    ds_sup = (ds_sup > err)? ds_sup : err;
}
printf("|S1 - S1_exact|_sup = %E \n", ds_sup);

printf("residual |A1 - U1*S1*V1**H|_F = %E \n", RnrmF[1] );

/* 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);

cudaDeviceReset();

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 I.

BIBLIOGRAPHY

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA 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

© 2014-2019 NVIDIA Corporation. All rights reserved.