CUSOLVER LIBRARY

DU-06709-001_v7.0 | March 2015

TABLE OF CONTENTS

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 $A \in R^{nxn}$, right-hand-side vector $b \in R^n$ and solution vector $x \in R^n$

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 a solve sparse linear system

$$Ax = b$$

and the least-squares problem

$$x = \mathrm{argmin}||A^* z - b||$$

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

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

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

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

1.3. cuSolverRF: Refactorization

The cuSolverRF library was designed to accelerate solution of sets of linear systems by fast re-factorization when given new coefficients in the same sparsity pattern

$$A_i x_i = f_i$$

where a sequence of coefficient matrices $A_i \in R^{nxn}$, right-hand-sides $f_i \in R^n$ and solutions $x_i \in R^n$ are given for **i=1,...,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,...,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:

cusolverDn<t><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. <**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:

cusolverSp[Host]<t>[<matrix data format>]<operation>[<output matrix data format>]<based on>

where **cuSolverSp** is the GPU path and **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 <**matrix data format**> is **csr**, compressed sparse row format.

The <**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

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

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.

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, 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 create CUDA streams using the function **`cudaStreamCreate()`** and set the stream to be used by each individual cuSolver library routine by calling for example **`cusolverDnSetStream()`** just before calling the actual cuSolverDN routine. Then, computations performed in separate streams would 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.

Chapter 3. CUSOLVER TYPES REFERENCE

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.

Value	Meaning
CUBLAS_FILL_MODE_LOWER	the lower part of the matrix is filled
CUBLAS_FILL_MODE_UPPER	the upper part of the matrix is filled

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.

Value	Meaning
CUBLAS_OP_N	the non-transpose operation is selected
CUBLAS_OP_T	the transpose operation is selected
CUBLAS_OP_C	the conjugate transpose operation is selected

3.1.4. 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 documenation 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.

CUSOLVER_STATUS_SUCCESS	The operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	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_ERROR	
	An internal cuSolver operation failed. This error is usually caused by a **cudaMemcpyAsync()** failure. **To correct:** check that the hardware, an appropriate version of the driver, and the cuSolver library are correctly installed. Also, check that the memory passed as a parameter to the routine is not being deallocated prior to the routine's completion.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	
	The matrix type is not supported by this function. This is usually caused by passing an invalid matrix descriptor to the function. **To correct:** check that the fields in **descrA** were set correctly.

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 **cusolverRfSetup()**, **cusolverRfSetupHost()**, **cusolverRfResetValues()**, **cusolveRfExtractBundledFactorsHost()** and **cusolverRfExtractSplitFactorsHost()** routines.

Value	Meaning
CUSOLVER_MATRIX_FORMAT_CSR	matrix format CSR is assumed. (default)
CUSOLVER_MATRIX_FORMAT_CSC	matrix format CSC is assumed.

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.

Value	Meaning
CUSOLVER_NUMERIC_BOOST_NOT_USED	numeric boosting not used. (default)
CUSOLVER_NUMERIC_BOOST_USED	numeric boosting used.

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.

Value	Meaning
CUSOLVER_RESET_VALUES_FAST_MODE_OFF	fast mode disabled. (default)
CUSOLVER_RESET_VALUES_FAST_MODE_ON	fast mode enabled.

3.3.5. cusolverRfFactorization_t

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

Value	Meaning
CUSOLVER_FACTORIZATION_ALG0	algorithm 0. (default)
CUSOLVER_FACTORIZATION_ALG1	algorithm 1.
CUSOLVER_FACTORIZATION_ALG2	algorithm 2. Domino-based scheme.

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.

Value	Meaning
CUSOLVER_TRIANGULAR_SOLVE_ALG0	algorithm 0.
CUSOLVER_TRIANGULAR_SOLVE_ALG1	algorithm 1. (default)
CUSOLVER_TRIANGULAR_SOLVE_ALG2	algorithm 2. Domino-based scheme.
CUSOLVER_TRIANGULAR_SOLVE_ALG3	algorithm 3. Domino-based scheme.

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 **cusolverRfSetup()**, **cusolverRfSetupHost()** and **cusolverRfExtractSplitFactorsHost()** routines.

Value	Meaning
CUSOLVER_UNIT_DIAGONAL_STORED_L	unit diagonal is stored in lower triangular factor. (default)
CUSOLVER_UNIT_DIAGONAL_STORED_U	unit diagonal is stored in upper triangular factor.
CUSOLVER_UNIT_DIAGONAL_ASSUMED_L	unit diagonal is assumed in lower triangular factor.
CUSOLVER_UNIT_DIAGONAL_ASSUMED_U	unit diagonal is assumed in upper triangular factor.

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

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

is represented as

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}$$

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 examle, the **m*n** (sub-)matrix

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ a_{2,1} & \cdots & a_{2,n} \\ \vdots & & \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$$

is represented as

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ a_{2,1} & \cdots & a_{2,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \\ \vdots & \ddots & \vdots \\ a_{\mathrm{lda},1} & \cdots & a_{\mathrm{lda},n} \end{pmatrix}$$

with its elements arranged linearly in memory as

$$(a_{1,1} \;\; a_{2,1} \;\; \cdots \;\; a_{m,1} \;\; \cdots \;\; a_{\mathrm{lda},1} \;\; \cdots \;\; a_{1,n} \;\; a_{2,n} \;\; \cdots \;\; a_{m,n} \;\; \cdots \;\; a_{\mathrm{lda},n})$$

where **lda** ≥ **m** is the leading dimension of **A**.

4.4. Matrix (CSR) Format

In CSR format the matrix is represented by the following parameters

parameter	type	size	Meaning
n	(int)		the number of rows (and columns) in the matrix.
nnz	(int)		the number of non-zero elements in the matrix.
csrRowPtr	(int *)	n+1	the array of offsets corresponding to the start of each row in the arrays csrColInd and csrVal. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix.
csrColInd	(int *)	nnz	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.**
csrVal	(S|D|C|Z)*	nnz	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.**

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 **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

$$\mathrm{csrRowPtr} = (0 \;\; 2 \;\; 4 \;\; 8 \;\; 9)$$

$$\text{csrColInd} = (0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 1 \quad 2 \quad 3 \quad 3)$$

$$\text{csrVal} = (1.0 \quad 3.0 \quad 4.0 \quad 6.0 \quad 2.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0)$$

4.5. Matrix (CSC) Format

In CSC format the matrix is represented by the following parameters

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

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

$$\text{cscColPtr} = (0 \quad 2 \quad 5 \quad 7 \quad 9)$$

$$\text{cscRowInd} = (0 \quad 2 \quad 0 \quad 1 \quad 2 \quad 1 \quad 2 \quad 2 \quad 3)$$

$$\text{cscVal} = (1.0 \quad 2.0 \quad 3.0 \quad 4.0 \quad 5.0 \quad 6.0 \quad 7.0 \quad 8.0 \quad 9.0)$$

Chapter 5. CUSOLVERDN: DENSE LAPACK FUNCTION REFERENCE

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.

parameter	Memory	In/out	Meaning
handle	host	output	the pointer to the handle to the cuSolverDN context.

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.

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.

Status Returned

CUSOLVER_STATUS_SUCCESS	the shutdown succeeded.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

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.

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
streamId	host	input	the stream to be used by the library.

Status Returned

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

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.

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
streamId	host	output	the stream to be used by the library.

Status Returned

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

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.

```
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.

```
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**.

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

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

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

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.

API of potrf

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
uplo	host	input	indicates if matrix **A** lower or upper part is stored, the other part is not referenced.
n	host	input	number of rows and columns of matrix **A**.
A	device	in/out	<type> array of dimension **lda * n** with **lda** is not less than **max(1,n)**.
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**.
Lwork	host	input	size of **Workspace**, returned by **potrf_bufferSize**.
devInfo	device	output	if **devInfo = 0**, the Cholesky factorization is successful. if **devInfo = -i**, the **i-th** parameter is wrong. if **devInfo = i**, the leading minor of order i is not positive definite.

Status Returned

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

5.2.2. cusolverDn<t>potrs()

```
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

$$A * X = B$$

where **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.

The user has to call **potrf** first to factorize matrix **A**. 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 **CUSBLAS_FILL_MODE_UPPER**, **A** is upper triangular Cholesky factor **U** corresponding to $A = U^* U^H$.

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.

API of potrs

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

Status Returned

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

5.2.3. cusolverDn<t>getrf()

These helper functions calculate the size of work buffers needed.

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

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

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

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.

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

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

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

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

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

This function computes the LU factorization of a **m×n** matrix

$$P*A = L*U$$

where **A** is a **m×n** matrix, **P** is a permutation matrix, **L** is a lower triangular matrix with unit diagonal, and **U** is an upper triangular matrix.

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

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

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

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

API of getrf

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
m	host	input	number of rows of matrix **A**.
n	host	input	number of columns of matrix **A**.
A	device	in/out	<type> array of dimension **lda * n** with **lda** is not less than **max(1,m)**.
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. if **devInfo = i**, the **U(i,i) = 0**.

Status Returned

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

5.2.4. 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

$$\mathrm{op(A)} * X = B$$

where **A** is a **n×n** matrix, and was LU-factored by **getrf**, that is, lower trianular part of A is **L**, and upper triangular part (including diagonal elements) of **A** is **U**. **B** is a **n×nrhs** right-hand side matrix.

The input parameter **trans** is defined by

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUBLAS\_OP\_N} \\ A^T & \text{if trans == CUBLAS\_OP\_T} \\ A^H & \text{if trans == CUBLAS\_OP\_C} \end{cases}$$

The input parameter **devIpiv** is an output of **getrf**. It contains pivot indices, which are used to permutate right-hand sides.

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
trans	host	input	operation **op(A)** that is non- or (conj.) transpose.
n	host	input	number of rows and columns of matrix **A**.
nrhs	host	input	number of right-hand sides.
A	device	input	<type> array of dimension **lda * n** with **lda** is not less than **max(1,n)**.
lda	host	input	leading dimension of two-dimensional array used to store matrix **A**.
devIpiv	device	input	array of size at least **n**, containing pivot indices.
B	device	output	<type> array of dimension **ldb * nrhs** with **ldb** is not less than **max(1,n)**.
ldb	host	input	leading dimension of two-dimensional array used to store matrix **B**.
devInfo	device	output	if **devInfo = 0**, the operation is successful. if **devInfo = -i**, the **i-th** parameter is wrong.

Status Returned

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

5.2.5. cusolverDn<t>geqrf()

These helper functions calculate the size of work buffers needed.

```
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.

```
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×n** matrix

$$A = Q * R$$

where **A** is a **m×n** matrix, **Q** is a **m×n** matrix, and **R** is a **n×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 **τ**. If **v** is original householder vector, **q** is the new householder vector corresponding to **τ**, satisying the following relation

$$I - 2 * v * v^H = I - \tau * q * q^H$$

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

API of geqrf

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
m	host	input	number of rows of matrix **A**.
n	host	input	number of columns of matrix **A**.
A	device	in/out	<type> array of dimension **lda * n** with **lda** is not less than **max(1,m)**.

lda	**host**	**input**	leading dimension of two-dimensional array used to store matrix **A**.
TAU	**device**	**output**	<type> array of dimension at least **min(m,n)**.
Workspace	**device**	**in/out**	working space, <type> array of size **Lwork**.
Lwork	**host**	**input**	size of working array **Workspace**.
devInfo	**device**	**output**	if **info = 0**, the LU factorization is successful. if **info = -i**, the **i-th** parameter is wrong.

Status Returned

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

5.2.6. cusolverDn<t>ormqr()

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.

```
cusolverStatus_t
cusolverDnCunmqr(cusolverDnHandle_t handle,
           cublasSideMode_t side,
           cublasOperation_t trans,
           int m,
           int n,
           int k,
           const cuComplex *A,
           int lda,
           const cuComplex *tau,
           cuComplex *C,
           int ldc,
           cuComplex *work,
           int lwork,
           int *devInfo);

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

This function overwrites **m×n** matrix **C** by

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

where **Q** is a unitary matrix formed by a sequence of elementary reflection vectors from QR factorization of **A**. Also for **Q**

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

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

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

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

parameter	Memory	In/out	Meaning

handle	**host**	**input**	handle to the cuSolverDN library context.
side	**host**	**input**	indicates if matrix **Q** is on the left or right of **C**.
trans	**host**	**input**	operation **op(Q)** that is non- or (conj.) transpose.
m	**host**	**input**	number of rows of matrix **A**.
n	**host**	**input**	number of columns of matrix **A**.
k	**host**	**input**	number of elementary relfections.
A	**device**	**in/out**	<type> 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.
lda	**host**	**input**	leading dimension of two-dimensional array used to store matrix **A**. if **side** is **CUBLAS_SIDE_LEFT**, lda >= max(1,m); if **side** is **CUBLAS_SIDE_RIGHT**, lda >= max(1,n).
tau	**device**	**output**	<type> 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.
C	**device**	**in/out**	<type> array of size **ldc * n**. On exit, **C** is overwritten by **op(Q)*C**.
ldc	**host**	**input**	leading dimension of two-dimensional array of matrix **C**. ldc >= max(1,m).
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.

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 wrong **lda** or **ldc**).
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>sytrf()

These helper functions calculate the size of the needed buffers.

```
cusolverStatus_t
cusolverDnSsytrf_bufferSize(cusolverDnHandle_t handle,
                      int n,
                      float *A,
                      int lda,
                      int *Lwork );

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

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

cusolverStatus_t
cusolverDnZsytrf_bufferSize(cusolverDnHandle_t handle,
                      int n,
                      cuDoubleComplex *A,
                      int lda,
                      int *Lwork );
```

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

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

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

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

```
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×n** symmetric indefinite matrix

A is a **n×n** symmetric matrix, only lower or upper part is meaningful. The input parameter **uplo** 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 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 **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 **work**. The input parameter **lwork** is size of the working space, and it is returned by **sytrf_bufferSize()**.

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

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

The output parameter **devIpiv** contains pivoting sequence. If **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 **uplo** is **CUSBLAS_FILL_MODE_UPPER** and **devIpiv(i-1) = devIpiv(i) = -m < 0**, **D(i-1:i,i-1:i)** is a 2x2 block, and **(i-1)-th** row/column is interchanged with **m-th** row/column. If **uplo** is **CUSBLAS_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

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
uplo	host	input	indicates if matrix **A** lower or upper part is stored, the other part is not referenced.
n	host	input	number of rows and columns of matrix **A**.
A	device	in/out	<type> array of dimension **lda * n** with **lda** is not less than **max(1,n)**.
lda	host	input	leading dimension of two-dimensional array used to store matrix **A**.
ipiv	device	output	array of size at least **n**, containing pivot indices.
work	device	in/out	working space, <type> array of size **lwork**.
lwork	host	input	size of working space **work**.
devInfo	device	output	if **devInfo = 0**, the LU factorization is successful. if **devInfo = -i**, the **i-th** parameter is wrong. if **devInfo = i**, the **D(i,i) = 0**.

Status Returned

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

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 function will calculate the size needed for work buffers in gebrd.

```
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 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 real **m×n** matrix **A** to upper or lower bidiagonal form **B** by an orthogonal transformation: $Q^H * A * P = B$

If **m>=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>=n**, the diagonal and the first superdiagonal are overwritten with the upper bidiagonal matrix **B**; the elements below the diagonal, with the array **TAUQ**, represent the orthogonal matrix **Q** as a product of elementary reflectors, and the elements above the first superdiagonal, with the array **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 **TAUQ**, represent the orthogonal matrix **Q** as a product of elementary reflectors, and the elements above the diagonal, with the array **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 **Work**. The input parameter **Lwork** is size of the working space, and it is returned by **gebrd_bufferSize()**.

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong.

Remark: **gebrd** only supports **m>=n**.

API of gebrd

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

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,n)).
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

5.3.2. cusolverDn<t>gesvd()

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

```
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. The rwork parameter is real valued, dimension 5*min(M,N). If info >0, rwork(1:min(M,N)) contains the unconverged superdiagonal elements of an upper bidiagonal matrix.

```
cusolverStatus_t
cusolverDnSgesvd (cusolverDnHandle_t handle,
            char jobu,
            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,
            char jobu,
            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. The rwork parameter is complex valued, dimension 5*min(M,N). If info >0,

rwork(1:min(M,N)) contains the unconverged superdiagonal elements of an upper bidiagonal matrix.

```
cusolverStatus_t
cusolverDnCgesvd (cusolverDnHandle_t handle,
            char jobu,
            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,
            char jobu,
            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×n** matrix **A** and corresponding the left and/or right singular vectors. The SVD is written

$$A = U * \Sigma * V^H$$

where **Σ** is an **m×n** matrix which is zero except for its **min(m,n)** diagonal elements, **U** is an **m×m** unitary matrix, and **V** is an **n×n** unitary matrix. The diagonal elements of **Σ** 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 **Work**. The input parameter **Lwork** is size of the working space, and it is returned by **gesvd_bufferSize()**.

If output parameter **devInfo = -i** (less than zero), the **i-th** parameter is wrong. if **bdsqr** did not converge, **devInfo** specifies how many superdiagonals of an intermediate bidiagonal form **B** did not converge to zero.

Note that the routine returns V^H, not **V**.

Remark 1: **gesvd** only supports **m>=n**.

Remark 2: **gesvd** only supports **jobu**='A' and **jobvt**='A' and returns matrix U and V^H.

API of gebrd

parameter	Memory	In/out	Meaning
handle	host	input	handle to the cuSolverDN library context.
jobu	host	input	specifies options for computing all or part of the matrix U: = 'A': all m columns of U are returned in array U: = 'S': the first min(m,n) columns of U (the left singular vectors) are returned in the array U; = 'O': the first min(m,n) columns of U (the left singular vectors) are overwritten on the array A; = 'N': no columns of U (no left singular vectors) are computed.
jobvt	host	input	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.
m	host	input	number of rows of matrix **A**.
n	host	input	number of columns of matrix **A**.
A	device	in/out	<type> array of dimension **lda * n** with **lda** is not less than **max(1,m)**. On exit, the contents of **A** are destroyed.
lda	host	input	leading dimension of two-dimensional array used to store matrix **A**.
S	device	output	<type> array of dimension **min(m,n)**. The singular values of A, sorted so that **S(i) >= S(i+1)**.
U	device	output	<type> array of dimension **ldu * m** with **ldu** is not less than **max(1,m)**. **U** contains the **m×m** unitary matrix **U**.
ldu	host	input	leading dimension of two-dimensional array used to store matrix **U**.
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	**host**	**input, needed for data types C,Z**	size of **Work**, returned by **gesvd_bufferSize**.
devInfo	**device**	**output**	if **devInfo = 0**, the operation is successful. if **devInfo = -i**, the **i-th** parameter is wrong. if **devInfo > 0**, **devInfo** indicates how many superdiagonals of an intermediate bidiagonal form **B** 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.

Chapter 6. CUSOLVERSP: SPARSE LAPACK FUNCTION REFERENCE

This chapter describes the API of cuSolverSP, which provides a subset of LAPACK funtions 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

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.

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

CUSOLVER_STATUS_SUCCESS	the shutdown succeeded.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

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

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

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 **issym** reports 1 if **A** is symmetric; otherwise, it reports 0.

The matrix **A** is an **m×m** sparse matrix that is defined in CSR storage format by the four arrays **csrValA**, **csrRowPtrA**, **csrEndPtrA** and **csrColIndA**.

The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.

The **csrlsvlu** and **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 **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, **issym** is undefined, and this state can only be detected by the return status code.

Input

parameter	MemorySpace	description
handle	host	handle to the cuSolverSP library context.
m	host	number of rows and columns of matrix **A**.
nnzA	host	number of nonzeros of matrix **A**. It is the size of **csrValA** and **csrColIndA**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrRowPtrA	host	integer array of **m** elements that contains the start of every row.
csrEndPtrA	host	integer array of **m** elements that contains the end of the last row plus one.
csrColIndA	host	integer array of **nnz**Acolumn indices of the nonzero elements of matrix **A**.

Output

parameter	MemorySpace	description
issym	host	1 if **A** is symmetric; 0 otherwise.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnzA<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

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

$$A*x = b$$

A is an **n×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**. **b** is the right-hand-side vector of size **n**, and **x** is the solution vector of size **n**.

The supported matrix type is **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 is wrong.

The linear system is solved by sparse LU with partial pivoting,

$$P*A = L*U$$

The performance of LU factorization mainly depends on zero fill-in. **cusolver** library provides **symrcm** to reduce zero fill-in. The input parameter **reorder** can enable **symrcm** if **reorder** is nonzero, otherwise, no reordering is performed.

If **reorder** is nonzero, **csrlsvlu** does

$$P*A*Q^T = L*U$$

where $Q = \text{symrcm}(A + A^T)$.

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

$$|U(j,j)| < \text{tol for some j}$$

The output parameter **singularity** is the smallest index of such **j**. If **A** is non-singular, **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 **singularity = 1** which means **U(1,1)≈0**.

Remark 1: **csrlsvlu** does traditional LU with partial pivoting, the pivot of k-th column is determined dynamically based on the k-th column of intermediate matrix. **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]). Before performing LU factorization, **csrlsvlu** over-estimates size of **L** and **U**, and allocates a buffer to contain factors **L** and **U**. There is no easy way to find a tight upper bound of size of LU, George and Ng [5] proves that sparsity pattern of cholesky factor of $A*A^T$ is a superset of sparsity pattern of **L** and **U**. Furthermore, they provides an algorithm to find sparisty pattern of QR factorization which is a superset of LU [6]. **csrlsvlu** uses QR factorization to estimate size of LU in the analysis phase. The cost of analysis phase is mainly on figuring out sparsity pattern of householder vectors in QR factorization. If system memory is insufficient to keep sparsity pattern of QR, **csrlsvlu** returns **CUSOLVER_STATUS_ALLOC_FAILED**. If the matrix is not banded, it is better to enable reordering to avoid **CUSOLVER_STATUS_ALLOC_FAILED**.

Remark 2: minimum degree ordering is the well-known technique to reduce zero fill-in of QR factorization. However in most cases, **symrcm** still performs well.

Remark 3: The user can reorder the matrix by other reordering scheme before calling **csrlsvlu** For example, **symrcm** is not a good choice on circuit simulation. The user can try column AMD.

Remark 4: if matrix **A** is singular and **reorder** is on, the output parameter **singularity** reports singularity of $P*A*P^T$.

Remark 5: only CPU (Host) path is provided.

Remark 6: multithreaded **csrlsvlu** is not avaiable yet. If QR does not incur much zero fill-in, **csrlsvqr** would be faster than **csrlsvlu**.

Input

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolverSP library context.
n	host	host	number of rows and columns of matrix **A**.
nnzA	host	host	number of nonzeros of matrix **A**.
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 **nnzA** (= **csrRowPtrA(n)** − **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	device	host	integer array of **n** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	device	host	integer array of **nnzA** (= **csrRowPtrA(n)** − **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
b	device	host	right hand side vector of size **n**.
tol	host	host	tolerance to decide if singular or not.
reorder	host	host	no ordering if **reorder**=0. Otherwise, **symrcm** is used to reduce zero fill-in.

Output

parameter	cusolverSp MemSpace	*Host MemSpace	description
x	device	host	solution vector of size **n**, x = inv(A)*b.
singularity	host	host	-1 if **A** is invertible. Otherwise, first index j such that **U(j,j)≈0**

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.

CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**n,nnzA<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.2.2. cusolverSp<t>csrlsvqr()

```
cusolverStatus_t
cusolverSpScsrlsvqr[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×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **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 **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 is 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.

$$|R(j,j)| < \text{tol for some j}$$

The output parameter **singularity** is the smallest index of such **j**. If **A** is non-singular, **singularity** is -1. The index is consistent with base index of **A**. For example, if 2nd column of **A** is the same as first column, then **A** is singular. If base index is 1, then **singularity = 2** which means **R(2,2)≈0**. If base index is 0, then **singularity = 1** which means **R(1,1)≈0**.

Remark: the parameter **reorder** has no effect because reordering is not implemented yet.

Input

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolverSP library context.
m	host	host	number of rows and columns of matrix **A**.
nnz	host	host	number of nonzeros of matrix **A**.
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 if singular or not.
reorder	host	host	no effect.

Output

parameter	cusolverSp MemSpace	*Host MemSpace	description
x	device	host	solution vector of size m, x = inv(A)*b.
singularity	host	host	-1 if A is invertible. Otherwise, first index j such that R(j,j)≈0

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (m,nnz<=0), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

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×m** symmetric postive definite sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **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 **CUSPARSE_MATRIX_TYPE_GENERAL** and upper triangular part of **A** is ignored. 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 $A=L+D+L^H$. Thereafter, **A** is assumed symmetric/Hermitian.

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 **singularity** has two meanings:

If **A** is not postive definite, there exists some integer **k** such that **A(0:k, 0:k)** is not positive definite. **singularity** is the minimum of such **k**.

If **A** is postive definite but near singular under tolerance (**max(tol,0)**), i.e. there exists some integer **k** such that $G(\mathrm{k},\mathrm{k}) <= \mathrm{tol}$. **singularity** is the minimum of such **k**.

Remark 1: **singularity** is base-0. If **A** is positive definite and not near singular under tolerance, **singularity** is -1.

Remark 2: if the user wants to know if **A** is postive definite or not, **tol=0** is enough.

Remark 3: reordering can greatly affect zero fill-in and have a huge performance impact. cuSolver does not provide reordering scheme, the user has to apply some reordering scheme to have decent performance, for example, **symrcm** or **symamd**.

Remark 4: the function works for in-place (**x** and **b** point to the same memory block) and out-of-place.

Remark 5: the function only works on 32-bit index, if matrix **G** has large zero fill-in such that number of nonzeros is bigger than 2^{31} , then **CUSOLVER_STATUS_ALLOC_FAILED** is returned.

Remark 6: the parameter **reorder** has no effect because reordering is not implemented yet.

Input

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolverSP library context.
m	host	host	number of rows and columns of matrix **A**.
nnz	host	host	number of nonzeros of matrix **A**.

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 effect.

Output

parameter	**cusolverSp MemSpace**	***Host MemSpace**	**description**
x	**device**	**host**	solution vector of size **m**, x = inv(A)*b.
singularity	**host**	**host**	-1 if **A** is symmetric postive definite.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

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

$$x = \text{argmin}||A*z - b||$$

A is an **m×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**. **b** is the right-hand-side vector of size **m**, and **x** is the least-square solution vector of size **n**.

The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. If **A** is square, symmetric/Hermitian and only lower/upper part is used or meaningful, the user has to extend the matrix into its missing upper/lower part, otherwise the result is wrong.

This function only works if **m** is greater or equal to **n**, in other words, **A** is a tall matrix.

The least-square problem is solved by sparse QR factorization with column pivoting,

$$A*P^T = Q*R$$

If **A** is of full rank (i.e. all columns of **A** are linear independent), then matrix **P** is an identity. Suppose rank of **A** is **k**, less than **n**, the permutation matrix **P** reorders columns of **A** in the following sense:

$$A*P^T = (A_1 \quad A_2) = (Q_1 \quad Q_2)\begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix}$$

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 **tol** decides numerical rank. The absolute value of every entry in R_{22} is less than or equal to **tolerance=max(tol,0)**.

The output parameter **rankA** denotes numerical rank of **A**.

Suppose $y = P^* x$ and $c = Q^H * b$, the least square problem can be reformed by

$$\min||A^* x - b|| = \min||R^* y - c||$$

or in matrix form

$$\begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

The output parameter **min_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

$$\begin{aligned} &\min||y|| \\ &\text{subject to} R_{11}^* y_1 + R_{12}^* y_2 = c_1 \end{aligned}$$

Or equivalently another least-square problem

$$\min||\begin{pmatrix} R_{11} \backslash R_{12} \\ I \end{pmatrix} * y_2 - \begin{pmatrix} R_{11} \backslash c_1 \\ O \end{pmatrix}||$$

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

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolver library context.
m	host	host	number of rows of matrix **A**.
n	host	host	number of columns of matrix **A**.
nnz	host	host	number of nonzeros of matrix **A**.
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 rank of **A**.

Output

parameter	**cusolverSp MemSpace**	***Host MemSpace**	**description**
rankA	**host**	**host**	numerical rank of **A**.
x	**device**	**host**	solution vector of size **n**, x=pinv(A)*b.
p	**device**	**host**	a vector of size **n**, which represents the permuation matrix **P** satisfying A*P^T=Q*R.
min_norm	**host**	**host**	**||A*x-b||, x=pinv(A)*b**.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.2.5. cusolverSp<t>csreigvsi()

```
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 maxiter,
                  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 maxiter,
                  double tol,
                  double *mu,
                  double *x);

cusolverStatus_t
cusolverSpCcsreigvsi[Host](cusolverSpHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  const cuComplex *csrValA,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  cuComplex mu0,
                  const cuComplex *x0,
                  int maxiter,
                  float tol,
                  cuComplex *mu,
                  cuComplex *x);

cusolverStatus_t
cusolverSpZcsreigvsi(cusolverSpHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  const cuDoubleComplex *csrValA,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  cuDoubleComplex mu0,
                  const cuDoubleComplex *x0,
                  int maxiter,
                  double tol,
                  cuDoubleComplex *mu,
                  cuDoubleComplex *x);
```

This function solves the simple eigenvalue problem $A*x = \lambda*x$ by shift-inverse method.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**. The output paramter **x** is the approximated eigenvector of size **m**,

The following shift-inverse method corrects eigenpair step-by-step until convergence.

It accepts several parameters:

mu0 is an initial guess of eigenvalue. The shift-inverse method will converge to the eigenvalue **mu** nearest **mu0** if **mu** is a singleton. Otherwise, the shift-inverse method may not converge.

x0 is an initial eigenvector. If the user has no preference, just chose **x0** randomly. **x0** must be nonzero. It can be non-unit length.

tol is the tolerance to decide convergence. If **tol** is less than zero, it would be treated as zero.

maxiter 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: initial eigenvalue	μ0	and initial vector x0
$x^{(0)}$	=	x0 of unit length
for j = 0 : maxiter		
solve	(A – μ0 * I) * $x^{(k+1)}$	= $x^{(k)}$
normalize	$x^{(k+1)}$	to unit length
compute approx. eigenvalue μ	= x^{H}	* A * x where x = $x^{(k+1)}$
if || A*$x^{(k+1)}$	– μ * $x^{(k+1)}$	|| < tolerance, then stop
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 **mu0** 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 eigevalue even the eigenvalue is a singleton. The user has to extend **A** to complex numbre and call **[cu|h]solver[C|Z]csreigvsi** with **mu0** not on real axis.

Remark 2: the tolerance **tol** should not be smaller than |mu0|*eps, where eps is machine zero. Otherwise, shift-inverse may not converge because of small tolerance.

Input

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolver library context.
m	host	host	number of rows and columns of matrix **A**.

nnz	**host**	**host**	number of nonzeros of matrix **A**.
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**.
mu0	**host**	**host**	initial guess of eigenvalue.
x0	**device**	**host**	initial guess of eigenvector, a vecotr of size **m**.
maxiter	**host**	**host**	maximum iterations in shift-inverse method.
tol	**host**	**host**	tolerance for convergence.

Output

parameter	cusolverSp MemSpace	*Host MemSpace	description
mu	**host**	**host**	approximated eigenvalue nearest **mu0** under tolerance.
x	**device**	**host**	approximated eigenvector of size **m**.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.2.6. cusolverSp<t>csreigs()

```
cusolverStatus_t
solverspScsreigs[Host](cusolverSpHandle_t handle,
                int m,
                int nnz,
                const cusparseMatDescr_t descrA,
                const float *csrValA,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cuComplex left_bottom_corner,
                cuComplex right_upper_corner,
                int *num_eigs);

cusolverStatus_t
cusolverSpDcsreigs[Host](cusolverSpHandle_t handle,
                int m,
                int nnz,
                const cusparseMatDescr_t descrA,
                const double *csrValA,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cuDoubleComplex left_bottom_corner,
                cuDoubleComplex right_upper_corner,
                int *num_eigs);

cusolverStatus_t
cusolverSpCcsreigs[Host](cusolverSpHandle_t handle,
                int m,
                int nnz,
                const cusparseMatDescr_t descrA,
                const cuComplex *csrValA,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cuComplex left_bottom_corner,
                cuComplex right_upper_corner,
                int *num_eigs);

cusolverStatus_t
cusolverSpZcsreigs[Host](cusolverSpHandle_t handle,
                int m,
                int nnz,
                const cusparseMatDescr_t descrA,
                const cuDoubleComplex *csrValA,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cuDoubleComplex left_bottom_corner,
                cuDoubleComplex right_upper_corner,
                int *num_eigs);
```

This function computes number of algebraic eigenvalues in a given box **B** by contour integral

$$\text{number of algebraic eigenvalues in box B} = \frac{1}{2*\pi*\sqrt{-1}} \oint_C \frac{P'(z)}{P(z)} dz$$

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_botoom_corner**) and the other is right upper corner (input parameter **right_upper_corner**). **P(z)=det(A - z*I)** is the characteristic polynomial of **A**.

A is an **m×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

parameter	cusolverSp MemSpace	*Host MemSpace	description
handle	host	host	handle to the cuSolverSP library context.
m	host	host	number of rows and columns of matrix **A**.
nnz	host	host	number of nonzeros of matrix **A**.
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**.
left_bottom_corner	host	host	left bottom corner of the box.
right_upper_corner	host	host	right upper corner of the box.

Output

parameter	cusolverSp MemSpace	*Host MemSpace	description
num_eigs	host	host	number of algebraic eigenvalues in a box.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (m,nnz<=0), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.3. Low Level Function Reference

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

6.3.1. cusolverSpXcsrsymrcm()

```
cusolverStatus_t
cusolverSpXcsrsymrcmHost(cusolverSpHandle_t handle,
             int n,
             int nnzA,
             const cusparseMatDescr_t descrA,
             const int *csrRowPtrA,
             const int *csrColIndA,
             int *p);
```

This function implements Symmetric Reverse Cuthill-McKee permutation. It returns a permutation vector **p** such that **A(p,p)** would concentrate nonzeros to diagonal. This is equivalent to **symrcm** in MATLAB, however the result may not be the same because of different heuristics in the pseudoperipheral finder. The **cuSolverSP** library implements **symrcm** based on the following two papers:

E. Chuthill and J. McKee, reducing the bandwidth of sparse symmetric matrices, ACM '69 Proceedings of the 1969 24th national conference, Pages 157-172

Alan George, Joseph W. H. Liu, An Implementation of a Pseudoperipheral Node Finder, ACM Transactions on Mathematical Software (TOMS) Volume 5 Issue 3, Sept. 1979, Pages 284-295

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:

$$\mathrm{A(p,p)} = P * A * P^T$$

A is an **n×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**.

The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Internally **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

parameter	*Host MemSpace	description
handle	host	handle to the cuSolverSP library context.
n	host	number of rows and columns of matrix **A**.
nnzA	host	number of nonzeros of matrix **A**. It is the size of **csrValA** and **csrColIndA**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrRowPtrA	host	integer array of **n+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	host	integer array of **nnzA**column indices of the nonzero elements of matrix **A**.

Output

parameter	hsolver	description
p	host	permutation vector of size **n**.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**n,nnzA<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.3.2. cusolverSpXcsrperm()

```
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);

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×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

parameter	cusolverSp MemSpace	description
handle	host	handle to the cuSolver library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
nnzA	host	number of nonzeros of matrix **A**. It is the size of **csrValA** and **csrColIndA**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrRowPtrA	host	integer array of **m+1** elements that contains the start of every row and end of last row plus one of matrix **A**.
csrColIndA	host	integer array of **nnzA**column indices of the nonzero elements of matrix **A**.
p	host	left permutation vector of size **m**.
q	host	right permutation vector of size **n**.
map	host	integer array of **nnzA** indices. If the user wants to get relationship between **A** and **B**, **map** must be set **0:1:(nnzA-1)**.
pBuffer	host	buffer allocated by the user, the size is returned by **csrperm_bufferSize()**.

Output

parameter	hsolver	description
csrRowPtrA	host	integer array of **m+1** elements that contains the start of every row and end of last row plus one of matrix **B**.

csrColIndA	**host**	integer array of **nnzA**column indices of the nonzero elements of matrix **B**.
map	**host**	integer array of **nnzA** indices that maps matrix **A** to matrix **B**.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSOLVER_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnzA<=0**), base index is not 0 or 1.
CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

6.3.3. cusolverSpXcsrqrBatched()

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

```
cusolverStatus_t
cusolverSpCreateCsrqrInfo(csrqrInfo_t *info);

cusolverStatus_t
cusolverSpDestroyCsrqrInfo(csrqrInfo_t info);
```

Analysis is the same for all data types, but each data type has a unique buffer size.

```
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);

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);

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.

```
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);

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

cusolverStatus_t
cusolverSpDcsrqrsvBatched(cusolverSpHandle_t handle,
                          int m,
                          int n,
                          int 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

$$x_j = \text{argmin}||A_j * z - b_j||, \text{ j } = 1,2,..., \text{ batchSize}$$

or a set of linear systems

$$A_j x_j = b_j, \text{ j } = 1,2,..., \text{ batchSize}$$

where each A_j is a **m×n** sparse matrix that is defined in CSR storage format by the four arrays **csrValA**, **csrRowPtrA** and **csrColIndA**.

The supported matrix type is **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,..., batchSize** . All matrices have the same sparity pattern, so only one copy of **csrRowPtrA** and **csrColIndA** is used. But the array **csrValA** stores coefficients of A_j one after another. In other words, **csrValA[k*nnzA : (k+1)*nnzA]** is the value of A_k .

The batched QR uses opaque data structure **csrqrInfo** to keep intermediate data, for example, matrix **Q** and matrix **R** of QR factorization. The user needs to create **csrqrInfo** first by **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]csrqrBufferInfoBatched**. The internal memory block is allocated implicitly during first call of **cusolverSp[S|D|C|Z]csrqrsvBatched**. The user only needs to allocate working space for **cusolverSp[S|D|C|Z]csrqrsvBatched**.

Instead of trying all batched matrices, the user can find maximum **batchSize** by querying **cusolverSp[S|D|C|Z]csrqrBufferInfoBatched**. For example, the user can increase **batchSize** till summation of **internalDataInBytes** and **workspaceInBytes** is greater than size of available device memory.

Suppose that the user needs to perform 253 linear solvers and available device memory is 2GB. if **cusolverSp[S|D|C|Z]csrqrsvBatched** can only afford **batchSize** 100, the user has to call **cusolverSp[S|D|C|Z]csrqrsvBatched** three times to finish all. The user calls **cusolverSp[S|D|C|Z]csrqrBufferInfoBatched** with **batchSize** 100. The opaque **info** would remember this **batchSize** and any subsequent call of **cusolverSp[S|D|C|Z]csrqrsvBatched** cannot exceed this value. In this example, the first two calls of **cusolverSp[S|D|C|Z]csrqrsvBatched** will use **batchSize** 100, and last call of **cusolverSp[S|D|C|Z]csrqrsvBatched** will use **batchSize** 53.

Example: suppose that A0, A1, .., A9 have the same sparsity pattern, the following code solves 10 linear systems $A_j x_j = b_j$, j = 0,2,..., 9 by batched sparse QR.

```
// Suppose that A0, A1, .., A9 are m x m sparse matrix represented by CSR
 format,
// Each matrix Aj has nonzero nnzA, and shares the same csrRowPtrA and
 csrColIndA.
// csrValA is aggregation of A0, A1, ..., A9.
int m ; // number of rows and columns of each Aj
int nnzA ; // number of nonzeros of each Aj
int *csrRowPtrA ; // each Aj has the same csrRowPtrA
int *csrColIndA ; // each Aj has the same csrColIndA
double *csrValA ; // aggregation of A0,A1,...,A9
cont int batchSize = 10; // 10 linear systems

cusolverSpHandle_t handle; // handle to cusolver library
csrqrInfo_t info = NULL;
cusparseMatDescr_t descrA = NULL;
void *pBuffer = NULL; // working space for numerical factorization

// step 1: create a descriptor
cusparseCreateMatDescr(&descrA);
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE); // A is base-1
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL); // A is a general
 matrix

// step 2: create empty info structure
cusolverSpCreateCsrqrInfo(&info);

// 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 Aj.
n	host	number of columns of each matrix Aj.
nnzA	host	number of nonzeros of each matrix Aj. It is the size csrColIndA.
descrA	host	the descriptor of each matrix Aj. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.
csrValA	device	<type> array of nnzA*batchSize nonzero elements of matrices A0, A1, ... 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 nnzAcolumn indices of the nonzero elements of each matrix Aj.
b	device	<type> array of m*batchSize of right-hand-side vectors b0, b1, ... 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 cusolverSpXcsrqrBufferInfoBatched().

Output

parameter	cusolverSp MemSpace	description
x	device	<type> array of m*batchSize of solution vectors x0, x1, ... 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

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,nnzA<=0), base index is not 0 or 1.

CUSOLVER_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSOLVER_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

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.

parameter	MemSpace	In/out	Meaning
handle	host	input	the handle to the cuSolverRF library.
nnzM	host	output	the number of non-zero elements of matrix **M**.
Mp	device	output	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**.
Mi	device	output	the array of column indices corresponding to the non-zero elements in the matrix **M**. It is assumed that this array is sorted by

			row and by column within each row. The array size is **nnzM**.
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

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

7.2. cusolverRfAnalyze()

```
cusolverStatus_t
cusolverRfAnalyze(cusolverRfHandle_t handle);
```

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

$$A = L * U$$

It is assumed that a prior call to the **cusolverRfSetup[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

$$A_i x_i = f_i$$

parameter	**MemSpace**	**In/out**	**Meaning**
handle	**host**	**in/out**	the handle to the cuSolverRF library.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

7.3. cusolverRfSetup()

```
cusolverStatus_t
cusolverRfSetup(/* 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

$$A_i x_i = f_i$$

The permutations **P** and **Q** represent the final composition of all the left and right reorderings applied to the original matrix **A**, respectively. However, these permutations are often associated with partial pivoting and reordering to minimize fill-in, respectively.

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

$$A_i x_i = f_i$$

parameter	MemSpace	In/out	Meaning
n	host	input	the number of rows (and columns) of matrix **A**.
nnzA	host	input	the number of non-zero elements of matrix **A**.
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**.

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	**device**	**input**	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**.
nnzL	**host**	**input**	the number of non-zero elements of matrix **L**.
csrRowPtrL	**device**	**input**	the array of offsets corresponding to the start of each row in the arrays **csrColIndL** and **csrValL**. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix **L**. The array size is **n+1**.
csrColIndL	**device**	**input**	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**.
csrValL	**device**	**input**	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**.
nnzU	**host**	**input**	the number of non-zero elements of matrix **U**.
csrRowPtrU	**device**	**input**	the array of offsets corresponding to the start of each row in the arrays **csrColIndU** and **csrValU**. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix **U**. The array size is **n+1**.
csrColIndU	**device**	**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**.
csrValU	**device**	**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**.
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 GLU library.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

7.4. cusolverRfSetupHost()

```
cusolverStatus_t
cusolverRfSetupHost(/* Input (in the host memory) */
                    int n,
                    int nnzA,
                    int* h_csrRowPtrA,
                    int* h_csrColIndA,
                    double* h_csrValA,
                    int nnzL,
                    int* h_csrRowPtrL,
                    int* h_csrColIndL,
                    double* h_csrValL,
                    int nnzU,
                    int* h_csrRowPtrU,
                    int* h_csrColIndU,
                    double* h_csrValU,
                    int* h_P,
                    int* h_Q,
                    /* Output */
                    cusolverRfHandle_t handle);
```

This routine assembles the internal data structures of the cuSolverRF library. It is often the first routine to be called after the call to the **cusolverRfCreate()** routine.

This routine accepts as input (on the host) the original matrix **A**, the lower **(L)** and upper **(U)** triangular factors, as well as the left **(P)** and the right **(Q)** permutations resulting from the full LU factorization of the first **(i=1)** linear system

$$A_i x_i = f_i$$

The permutations **P** and **Q** represent the final composition of all the left and right reorderings applied to the original matrix **A**, respectively. However, these permutations are often associated with partial pivoting and reordering to minimize fill-in, respectively.

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

$$A_i x_i = f_i$$

parameter	MemSpace	In/out	Meaning

n	**host**	**input**	the number of rows (and columns) of matrix **A**.
nnzA	**host**	**input**	the number of non-zero elements of matrix **A**.
h_csrRowPtrA	**host**	**input**	the array of offsets corresponding to the start of each row in the arrays **h_csrColIndA** and **h_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**.
h_csrColIndA	**host**	**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**.
h_csrValA	**host**	**input**	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**.
nnzL	**host**	**input**	the number of non-zero elements of matrix **L**.
h_csrRowPtrL	**host**	**input**	the array of offsets corresponding to the start of each row in the arrays **h_csrColIndL** and **h_csrValL**. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix **L**. The array size is **n+1**.
h_csrColIndL	**host**	**input**	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**.
h_csrValL	**host**	**input**	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**.
nnzU	**host**	**input**	the number of non-zero elements of matrix **U**.
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

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

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.

parameter	MemSpace	In/out	Meaning
handle	**host**	**output**	the pointer to the cuSolverRF library handle.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

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.

parameter	MemSpace	In/out	Meaning
handle	host	input	the handle to the cuSolverRF library.
h_nnzM	host	output	the number of non-zero elements of matrix **M**.
h_Mp	host	output	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**.
h_Mi	host	output	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**.
h_Mx	host	output	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**.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

7.7. cusolverRfExtractSplitFactorsHost()

```
cusolverStatus_t
cusolverRfExtractSplitFactorsHost(/* Input */
                                  cusolverRfHandle_t handle,
                                  /* Output (in the host memory) */
                                  int* h_nnzL,
                                  int** h_Lp,
                                  int** h_Li,
                                  double** h_Lx,
                                  int* h_nnzU,
                                  int** h_Up,
                                  int** h_Ui,
                                  double** h_Ux);
```

This routine extracts lower **(L)** and upper **(U)** triangular factors from the cuSolverRF library handle into the host memory. It is assumed that a prior call to the **cusolverRfRefactor()** was done in order to generate these triangular factors.

parameter	MemSpace	In/out	Meaning
handle	host	input	the handle to the cuSolverRF library.
h_nnzL	host	output	the number of non-zero elements of matrix L.
h_Lp	host	output	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.
h_Li	host	output	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.
h_Lx	host	output	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.
h_nnzU	host	output	the number of non-zero elements of matrix U.
h_Up	host	output	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.
h_Ui	host	output	the array of column indices corresponding to the non-zero elements in the matrix U. It is assumed that this array is sorted by

			row and by column within each row. The array size is **h_nnzU**.
h_Ux	**host**	**output**	the array of values corresponding to the non-zero elements in the matrix **U**. It is assumed that this array is sorted by row and by column within each row. The array size is **h_nnzU**.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

7.8. cusolverRfDestroy()

```
cusolverStatus_t cusolverRfDestroy(cusolverRfHandle_t handle);
```

This routine shuts down the cuSolverRF library. It releases acquired resources and must be called after all the cuSolverRF library routines.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the cuSolverRF library handle.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.9. cusolverRfGetMatrixFormat()

```
cusolverStatus_t
cusolverRfGetMatrixFormat(cusolverRfHandle_t handle,
                          cusolverRfMatrixFormat_t *format,
                          cusolverRfUnitDiagonal_t *diag);
```

This routine gets the matrix format used in the **cusolverRfSetup()**, **cusolverRfSetupHost()**, **cusolverRfResetValues()**, **cusolverRfExtractBundledFactorsHost()** and **cusolverRfExtractSplitFactorsHost()** routines.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
format	**host**	**output**	the enumerated matrix format type.

diag	**host**	**output**	the enumerated unit diagonal type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.10. cusolverRfGetNumericProperties()

```
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**.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
zero	**host**	**output**	the value below which zero pivot is flagged.
boost	**host**	**output**	the value which is substituted for zero pivot (if the later is flagged).

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.11. cusolverRfGetNumericBoostReport()

```
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.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
report	**host**	**output**	the enumerated boosting report type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.12. cusolverRfGetResetValuesFastMode()

```
cusolverStatus_t
cusolverRfGetResetValuesFastMode(cusolverRfHandle_t handle,
                                 rfResetValuesFastMode_t *fastMode);
```

This routine gets the mode used in the **cusolverRfResetValues** routine.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
fastMode	**host**	**output**	the enumerated mode type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

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()**.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
alg	**host**	**output**	the enumerated algorithm type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

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 paralellism.

This routine may be called multiple times, once for each of the linear systems

$$A_i x_i = f_i$$

parameter	Memory	In/out	Meaning
handle	host	in/out	the handle to the cuSolverRF library.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_ZERO_PIVOT	a zero pivot was encountered during the computation.

7.15. cusolverRfResetValues()

```
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]** 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

$$A_i x_i = f_i$$

parameter	MemSpace	In/out	Meaning

n	**host**	**input**	the number of rows (and columns) of matrix **A**.
nnzA	**host**	**input**	the number of non-zero elements of matrix **A**.
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**.
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	**device**	**input**	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**.
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

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

7.16. cusolverRfSetMatrixFormat()

```
cusolverStatus_t
cusolverRfSetMatrixFormat(cusolverRfHandle_t handle,
                          gluMatrixFormat_t format,
                          gluUnitDiagonal_t diag);
```

This routine sets the matrix format used in the **cusolverRfSetup()**, **cusolverRfSetupHost()**, **cusolverRfResetValues()**, **cusolverRfExtractBundledFactorsHost()** and **cusolverRfExtractSplitFactorsHost()** routines. It may be called once prior to **cusolverRfSetup()** and **cusolverRfSetupHost()** routines.

parameter	MemSpace	In/out	Meaning

handle	**host**	**input**	the handle to the cuSolverRF library.
format	**host**	**input**	the enumerated matrix format type.
diag	**host**	**input**	the enumerated unit diagonal type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an enumerated mode parameter is wrong.

7.17. cusolverRfSetNumericProperties()

```
cusolverStatus_t
cusolverRfSetNumericProperties(cusolverRfHandle_t handle,
                               double zero,
                               double boost);
```

This routine sets the numeric values used for checking for "zero" pivot and for boosting it in the **cusolverRfRefactor()** and **cusolverRfSolve()** routines. It may be called multiple times prior to **cusolverRfRefactor()** and **cusolverRfSolve()** routines. The numeric boosting will be used only if **boost > 0.0**.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
zero	**host**	**input**	the value below which zero pivot is flagged.
boost	**host**	**input**	the value which is substituted for zero pivot (if the later is flagged).

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.18. cusolverRfSetResetValuesFastMode()

```
cusolverStatus_t
cusolverRfSetResetValuesFastMode(cusolverRfHandle_t handle,
                                 gluResetValuesFastMode_t fastMode);
```

This routine sets the mode used in the **cusolverRfResetValues** routine. The fast mode requires extra memory and is recommended only if very fast calls to **cusolverRfResetValues()** are needed. It may be called once prior to **cusolverRfAnalyze()** routine.

parameter	MemSpace	In/out	Meaning
handle	host	input	the handle to the cuSolverRF library.
fastMode	host	input	the enumerated mode type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an enumerated mode parameter is wrong.

7.19. cusolverRfSetAlgs()

```
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.

parameter	MemSpace	In/out	Meaning
handle	host	input	the handle to the cuSolverRF library.
alg	host	input	the enumerated algorithm type.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

7.20. cusolverRfSolve()

```
cusolverStatus_t
cusolverRfSolve(/* Input (in the device memory) */
          cusolverRfHandle_t handle,
          int *P,
          int *Q,
          int nrhs,
          double *Temp,
          int ldt,
          /* Input/Output (in the device memory) */
          double *XF,
          /* Input */
          int ldxf);
```

This routine performs the forward and backward solve with the lower $L \in R^{nxn}$ and upper $U \in R^{nxn}$ triangular factors resulting from the LU re-factorization

$$A = L * U$$

which is assumed to have been computed by a prior call to the **cusolverRfRefactor()** routine.

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

$$AX = (\mathrm{LU})X = L(\mathrm{UX}) = LY = F \text{ where } UX = Y$$

even though currently only a single rhs is supported.

This routine may be called multiple times, once for each of the linear systems

$$A_i x_i = f_i$$

parameter	MemSpace	In/out	Meaning
handle	host	output	the handle to the cuSolverRF library.
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.
nrhs	host	input	the number right-hand-sides to be solved.
Temp	host	input	the dense matrix that contains temporary workspace (of size ldt*nrhs).
ldt	host	input	the leading dimension of dense matrix Temp (ldt >= n).
XF	host	in/out	the dense matrix that contains the right-hand-sides F and solutions X (of size ldxf*nrhs).
ldxf	host	input	the leading dimension of dense matrix XF (ldxf >= n).

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

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

$$A_j x_j = b_j, \text{ j } = 1,2,..., \text{ batchSize}$$

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_1.

This routine accepts as input (on the host) the original matrix **A** (sparsity pattern and batched values), the lower **(L)** and upper **(U)** triangular factors, as well as the left **(P)** and the right **(Q)** permutations resulting from the full LU factorization of the first **(i=1)** linear system

$$A_i x_i = f_i$$

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

$$A_i x_i = f_i$$

parameter	MemSpace	In/out	Meaning
batchSize	host	input	the number of matrices in the batched mode.
n	host	input	the number of rows (and columns) of matrix **A**.
nnzA	host	input	the number of non-zero elements of matrix **A**.
h_csrRowPtrA	host	input	the array of offsets corresponding to the start of each row in the arrays **h_csrColIndA** and **h_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**.
h_csrColIndA	host	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**.
h_csrValA_array	host	input	array of pointers of size **batchSize**, each pointer points to the array of values corresponding to the non-zero elements in the matrix.
nnzL	host	input	the number of non-zero elements of matrix **L**.
h_csrRowPtrL	host	input	the array of offsets corresponding to the start of each row in the arrays **h_csrColIndL** and **h_csrValL**. This array has also an extra entry at the end that stores the number of non-zero elements in the matrix **L**. The array size is **n+1**.
h_csrColIndL	host	input	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**.
h_csrValL	host	input	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**.
nnzU	host	input	the number of non-zero elements of matrix **U**.

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

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

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

$$A_j x_j = b_j, \text{ j } = 1,2,..., \text{ batchSize}$$

parameter	Memory	In/out	Meaning

handle	**host**	**in/out**	the handle to the cuSolverRF library.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_ALLOC_FAILED	an allocation of memory failed.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

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

$$A_j x_j = b_j, \text{ j } = 1,2,..., \text{batchSize}$$

The input parameter **csrValA_array** is an array of pointers on device memory. **csrValA_array(j)** points to matrix A_j which is also on device memory.

parameter	MemSpace	In/out	Meaning
batchSize	**host**	**input**	the number of matrices in batched mode.
n	**host**	**input**	the number of rows (and columns) of matrix **A**.
nnzA	**host**	**input**	the number of non-zero elements of matrix **A**.
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**.
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

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

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 paralellism.

Remark: **cusolverRfBatchRefactor()** would not report any failure of LU refactorization. The user has to call **cusolverRfBatchZeroPivot()** to know which matrix failed the LU refactorization.

parameter	**Memory**	**In/out**	**Meaning**
handle	**host**	**in/out**	the handle to the cuSolverRF library.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.

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:

$z_j = P * b_j$

$M_j * y_j = z_j$

$x_j = Q^T * y_j$

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.

parameter	Memory	In/out	Meaning
handle	host	output	the handle to the cuSolverRF library.
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.
nrhs	host	input	the number right-hand-sides to be solved.
Temp	host	input	the dense matrix that contains temporary workspace (of size ldt*nrhs).
ldt	host	input	the leading dimension of dense matrix Temp (ldt >= n).
XF_array	host	in/out	array of pointers of size batchSize, each pointer points to the dense matrix

			that contains the right-hand-sides **F** and solutions **X** (of size **ldxf*nrhs**).
ldxf	**host**	**input**	the leading dimension of dense matrix **XF** (**ldxf >= n**).

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSOLVER_STATUS_INVALID_VALUE	an unsupported value or parameter was passed.
CUSOLVER_STATUS_EXECUTION_FAILED	a kernel failed to launch on the GPU.
CUSOLVER_STATUS_INTERNAL_ERROR	an internal operation failed.

7.26. cusolverRfBatchZeroPivot()

```
cusolverStatus_t
cusolverRfBatchZeroPivot(/* Input */
                    cusolverRfHandle_t handle
                    /* Output (in the host memory) */
                    int *position);
```

Although A_j is close to each other, it does not mean $M_j = P * A_j * Q^T = L_j * U_j$ exists for every j. The user can query which matrix failed LU refactorization by checking corresponding value in **position** array. The input parameter **position** is an integer array of size **batchSize**.

The **j-th** component denotes the refactorization result of matrix A_j. If **position(j)** is -1, the LU refactorization of matrix A_j is successful. If **position(j)** is **k >= 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.

parameter	MemSpace	In/out	Meaning
handle	**host**	**input**	the handle to the cuSolverRF library.
position	**host**	**output**	integer array of size **batchSize**. The value of **position(j)** reports singularity of matrix **Aj**, -1 if no structural/ numerical zero, **k >= 0** if **Aj(k,k)** is either structural zero or numerical zero.

Status Returned

CUSOLVER_STATUS_SUCCESS	the operation completed successfully.
CUSOLVER_STATUS_NOT_INITIALIZED	the library was not initialized.

CUSOLVER_STATUS_ZERO_PIVOT	a zero pivot was encountered during the computation.

Appendix A. CUSOLVERRF EXAMPLES

A.1. cuSolverRF In-memory Example

This is an example in the C programming language of how to use the standard routines in the cuSolverRF library. We focus on solving the set of linear systems

$$A_i x_i = f_i$$

but we change the indexing from one- to zero-based to follow the C programming language. The example begins with the usual includes and main()

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cusolverRf.h"

#define TEST_PASSED  0
#define TEST_FAILED  1

int main (void){
    /* matrix A */
    int n;
    int nnzA;
    int *Ap=NULL;
    int *Ai=NULL;
    double *Ax=NULL;
    int *d_Ap=NULL;
    int *d_Ai=NULL;
    double *d_rAx=NULL;
    /* matrices L and U */
    int nnzL, nnzU;
    int *Lp=NULL;
    int *Li=NULL;
    double* Lx=NULL;
    int *Up=NULL;
    int *Ui=NULL;
    double* Ux=NULL;
    /* reordering matrices */
    int *P=NULL;
    int *Q=NULL;
    int * d_P=NULL;
    int * d_Q=NULL;
    /* solution and rhs */
    int nrhs; //# of rhs for each system (currently only =1 is supported)
    double *d_X=NULL;
    double *d_T=NULL;
    /* cuda */
    cudaError_t cudaStatus;
    /* cuolverRf */
    cusolverRfHandle_t gH=NULL;
    cusolverStatus_t status;
    /* host sparse direct solver */
    /* ... */
    /* other variables */
    int tnnzL, tnnzU;
    int *tLp=NULL;
    int *tLi=NULL;
    double *tLx=NULL;
    int *tUp=NULL;
    int *tUi=NULL;
    double *tUx=NULL;
    double t1, t2;
```

Then we initialize the library.

```
   /* ASSUMPTION: recall that we are solving a set of linear systems
      A_{i} x_{i} = f_{i}  for i=0,...,k-1
      where the sparsity pattern of the coefficient matrices A_{i}
      as well as the reordering to minimize fill-in and the pivoting
      used during the LU factorization remain the same. */


   /* Step 1: solve the first linear system (i=0) on the host,
              using host sparse direct solver, which involves
              full LU factorization and solve. */
   /* ... */


   /* Step 2: interface to the library by extracting the following
              information from the first solve:
              a) triangular factors L and U
              b) pivoting and reordering permutations P and Q
              c) also, allocate all the necessary memory */
   /* ... */


   /* Step 3: use the library to solve subsequent (i=1,...,k-1) linear systems
   a) the  library setup (called only once) */
   //create handle
   status = cusolverRfCreate(&gH);
   if (status != CUSOLVER_STATUS_SUCCESS){
       printf ("[cusolverRf status \%d]\n",status);
       return TEST_FAILED;
   }

   //set fast mode
   status =
cusolverRf_set_reset_values_fast_mode(gH,GLU_RESET_VALUES_FAST_MODE_ON);
   if (status != CUSOLVER_STATUS_SUCCESS){
       printf ("[cusolverRf status \%d]\n",status);
       return TEST_FAILED;
   }
```

Call refactorization and solve.

```
   //assemble internal data structures (you should use the coeffcient matrix A
   //corresponding to the second (i=1) linear system in this call)
   t1 = cusolver_test_seconds();
   status = cusolverRfSetupHost(n, nnzA, Ap, Ai, Ax,
                                nnzL, Lp, Li, Lx, nnzU, Up, Ui, Ux, P, Q, gH);
   cudaStatus = cudaDeviceSynchronize();
   t2 = cusolver_test_seconds();
   if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
       printf ("[cusolverRf status \%d]\n",status);
       return TEST_FAILED;
   }
   printf("cusolverRfSetupHost time = \%f (s)\n", t2-t1);

   //analyze available parallelism
   t1 = cusolver_test_seconds();
   status = cusolverRfAnalyze(gH);
   cudaStatus = cudaDeviceSynchronize();
   t2 = cusolver_test_seconds();
   if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
       printf ("[cusolverRf status \%d]\n",status);
       return TEST_FAILED;
   }
   printf("cusolverRfAnalyze time = \%f (s)\n", t2-t1);

   /* b) The  library subsequent (i=1,...,k-1) LU re-factorization
         and solve (called multiple times). */
   for (i=1; i<k; i++){
       //LU re-factorization
       t1 = cusolver_test_seconds();
       status = cusolverRfRefactor(gH);
       cudaStatus = cudaDeviceSynchronize();
       t2 = cusolver_test_seconds();
       if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess))
{
           printf ("[cusolverRF status \%d]\n",status);
           return TEST_FAILED;
       }
       printf("cuSolverReRefactor time = \%f (s)\n", t2-t1);

       //forward and backward solve
       t1 = cusolver_test_seconds();
       status = cusolverRfSolve(gH, d_P, d_Q, nrhs, d_T, n, d_X, n);
       cudaStatus = cudaDeviceSynchronize();
       t2 = cusolver_test_seconds();
       if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess))
{
           printf ("[cusolverRf status \%d]\n",status);
           return TEST_FAILED;
       }
       printf("cusolverRfSolve time = \%f (s)\n", t2-t1);
```

Extract the results and return.

```
        // extract the factors (if needed)
        status = cusolverRf_extract_split_factors_host(gH, &tnnzL, &tLp, &tLi,
 &tLx,
                                                       &tnnzU, &tUp, &tUi, &tUx);
        if(status != CUSOLVER_STATUS_SUCCESS){
            printf ("[cusolverRf status \%d]\n",status);
            return TEST_FAILED;
        }
        /*
        //print
        int row, j;
        printf("printing L\n");
        for (row=0; row<n; row++){
            for (j=tLp[row]; j<tLp[row+1]; j++){
                printf("\%d,\%d,\%f\n",row,tLi[j],tLx[j]);
            }
        }
        printf("printing U\n");
        for (row=0; row<n; row++){
            for (j=tUp[row]; j<tUp[row+1]; j++){
                printf("\%d,\%d,\%f\n",row,tUi[j],tUx[j]);
            }
        }
        */

        /* perform any other operations based on the solution */
        /* ... */

        /* check if done */
        /* ... */

        /* proceed to solve the next linear system */
        // update the coefficient matrix using reset values
        // (assuming that the new linear system, in other words,
        //  new values are already on the GPU in the array d_rAx)
        t1 = cusolver_test_seconds();
        status = cusolverRf_reset_values(n,nnzA,d_Ap,d_Ai,d_rAx,d_P,d_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("cusolverRf_reset_values time = \%f (s)\n", t2-t1);
    }

    /* free memory and exit */
    /* ... */
    return TEST_PASSED;
}
```

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

```
#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 contiguosly 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 contiguosly 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

```
/* 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.

```
    //assemble internal data structures
    t1 = cusolver_test_seconds();
    status = cusolverRfBatch_SetupHost(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("cusolverRfbatch_assemble_host time = %f (s)\n", t2-t1);

    //analyze available parallelism
    t1 = cusolver_test_seconds();
    status = cusolverRfBatch_Analyze(gH);
    cudaStatus = cudaDeviceSynchronize();
    t2 = cusolver_test_seconds();
    if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
        printf ("[cusolverRf status %d]\n",status);
        return TEST_FAILED;
    }
    printf("cusolverRfbatch_analyze 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 = cusolverRfBatch_Refactor(gH);
    cudaStatus = cudaDeviceSynchronize();
    t2 = cusolver_test_seconds();
    if ((status != CUSOLVER_STATUS_SUCCESS) || (cudaStatus != cudaSuccess)) {
        printf ("[cusolverRf status %d]\n",status);
        return TEST_FAILED;
    }
    printf("cusolverRfbatch_refactor time = %f (s)\n", t2-t1);

    //forward and backward solve
    t1 = cusolver_test_seconds();
    status = cusolverRfBatch_Solve(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("cusolverRfBatch_Solve time = %f (s)\n", t2-t1);

    /* free memory and exit */
    /* ... */
    return TEST_PASSED;
}
```

Appendix B. CSR QR BATCH EXAMPLES

B.1. Batched Sparse QR example 1

This chapter provides a simple example in the C programming language of how to use batched sparse QR to solver a set of linear systems

$$A_i x_i = b_i$$

All matrices A_i are small perturbations of

$$A = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.1 & 0.1 & 0.1 & 4.0 \end{pmatrix}$$

All right-hand side vectors b_i are small perturbation of the Matlab vector 'ones(4,1)'.

We assume device memory is big enough to compute all matrices in one pass.

The usual includes and main definition

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <cusolver.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               |
 *  A = |       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

```
    const int m = 4 ;
    const int nnzA = 7;
    const int csrRowPtrA[m+1]  = { 1, 2, 3, 4, 8};
    const int csrColIndA[nnzA] = { 1, 2, 3, 1, 2, 3, 4};
    const double csrValA[nnzA] = { 1.0, 2.0, 3.0, 0.1, 0.1, 0.1, 4.0};
    const double b[m] = {1.0, 1.0, 1.0, 1.0};
    const int batchSize = 17;

    double *csrValABatch = (double*)malloc(sizeof(double)*nnzA*batchSize);
    double *bBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    double *xBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    assert( NULL != csrValABatch );
    assert( NULL != bBatch );
    assert( NULL != xBatch );

// step 1: prepare Aj and bj on host
//  Aj is a small perturbation of A
//  bj is a small perturbation of b
//  csrValABatch = [A0, A1, A2, ...]
//  bBatch = [b0, b1, b2, ...]
    for(int colidx = 0 ; colidx < nnzA ; colidx++){
        double Areg = csrValA[colidx];
        for (int batchId = 0 ; batchId < batchSize ; batchId++){
            double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
            csrValABatch[batchId*nnzA + colidx] = Areg + eps;
        }
    }

    for(int j = 0 ; j < m ; j++){
        double breg = b[j];
        for (int batchId = 0 ; batchId < batchSize ; batchId++){
            double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
            bBatch[batchId*m + j] = breg + eps;
        }
    }

// step 2: create cusolver handle, qr info and matrix descriptor
    cusolver_status = cusolverSpCreate(&cusolverH);
    assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

    cusparse_status = cusparseCreateMatDescr(&descrA);
    assert(cusparse_status == CUSPARSE_STATUS_SUCCESS);

    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
    cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE); // base-1

    cusolver_status = cusolverSpCreateCsrqrInfo(&info);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);
```

Call the solver

```
// step 3: copy Aj and bj to device
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(double) * nnzA *
 batchSize);
    cudaStat2 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int) * nnzA);
    cudaStat3 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int) * (m+1));
    cudaStat4 = cudaMalloc ((void**)&d_b         , sizeof(double) * m *
 batchSize);
    cudaStat5 = cudaMalloc ((void**)&d_x         , sizeof(double) * m *
 batchSize);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);
    assert(cudaStat4 == cudaSuccess);
    assert(cudaStat5 == cudaSuccess);

    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 = cudaMalloc((void**)&buffer_qr, size_qr);
    assert(cudaStat1 == cudaSuccess);
```

Get results back

```
// 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

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cusolver.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

/*      | 1               |
 *  A = |       2         |
 *      |            3    |
 *      | 0.1  0.1  0.1  4 |
 *  CSR of A is based-1
 *
 *  b = [1 1 1 1]
 */
```

Create the library handle

```
    const int m = 4 ;
    const int nnzA = 7;
    const int csrRowPtrA[m+1]  = { 1, 2, 3, 4, 8};
    const int csrColIndA[nnzA] = { 1, 2, 3, 1, 2, 3, 4};
    const double csrValA[nnzA] = { 1.0, 2.0, 3.0, 0.1, 0.1, 0.1, 4.0};
    const double b[m] = {1.0, 1.0, 1.0, 1.0};
    const int batchSize = 17;

    double *csrValABatch = (double*)malloc(sizeof(double)*nnzA*batchSize);
    double *bBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    double *xBatch       = (double*)malloc(sizeof(double)*m*batchSize);
    assert( NULL != csrValABatch );
    assert( NULL != bBatch );
    assert( NULL != xBatch );

// step 1: prepare Aj and bj on host
//  Aj is a small perturbation of A
//  bj is a small perturbation of b
//  csrValABatch = [A0, A1, A2, ...]
//  bBatch = [b0, b1, b2, ...]
    for(int colidx = 0 ; colidx < nnzA ; colidx++){
        double Areg = csrValA[colidx];
        for (int batchId = 0 ; batchId < batchSize ; batchId++){
            double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
            csrValABatch[batchId*nnzA + colidx] = Areg + eps;
        }
    }

    for(int j = 0 ; j < m ; j++){
        double breg = b[j];
        for (int batchId = 0 ; batchId < batchSize ; batchId++){
            double eps = ((double)((rand() % 100) + 1)) * 1.e-4;
            bBatch[batchId*m + j] = breg + eps;
        }
    }

// step 2: create cusolver handle, qr info and matrix descriptor
    cusolver_status = cusolverSpCreate(&cusolverH);
    assert (cusolver_status == CUSOLVER_STATUS_SUCCESS);

    cusparse_status = cusparseCreateMatDescr(&descrA);
    assert(cusparse_status == CUSPARSE_STATUS_SUCCESS);

    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
    cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE); // base-1

    cusolver_status = cusolverSpCreateCsrqrInfo(&info);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);
```

Set up the data

```
// step 3: copy Aj and bj to device
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(double) * nnzA *
 batchSize);
    cudaStat2 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int) * nnzA);
    cudaStat3 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int) * (m+1));
    cudaStat4 = cudaMalloc ((void**)&d_b         , sizeof(double) * m *
 batchSize);
    cudaStat5 = cudaMalloc ((void**)&d_x         , sizeof(double) * m *
 batchSize);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == cudaSuccess);
    assert(cudaStat3 == cudaSuccess);
    assert(cudaStat4 == cudaSuccess);
    assert(cudaStat5 == cudaSuccess);

// don't copy csrValABatch and bBatch because device memory may be big enough
    cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA,
 cudaMemcpyHostToDevice);
    cudaStat2 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (m+1),
 cudaMemcpyHostToDevice);
    assert(cudaStat1 == cudaSuccess);
    assert(cudaStat2 == 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,
            // d_csrValA is don't care
            descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
            batchSizeMax, // WARNING: use batchSizeMax
            info,
            &size_internal,
            &size_qr);
        assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

        if ( (size_internal + size_qr) > free_mem ){
            // current batchSizeMax exceeds hardware limit, so cut it by half.
            batchSizeMax /= 2; break;
        }
        batchSizeMax *= 2; // double batchSizMax and try it again.
    }
    // correct batchSizeMax such that it is not greater than batchSize.
    batchSizeMax = imin(batchSizeMax, batchSize);
    printf("batchSizeMax = %d\n", batchSizeMax);

// Assume device memory is not big enough, and batchSizeMax = 2
    batchSizeMax = 2;
```

Perform analysis and call solve

```
// step 6: prepare working space
// [necessary]
// Need to call cusolverDcsrqrBufferInfoBatched again with batchSizeMax
// to fix batchSize used in numerical factorization.
    cusolver_status = cusolverSpDcsrqrBufferInfoBatched(
         cusolverH, m, m, nnzA,
         // d_csrValA is don't care
         descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA,
         batchSizeMax, // WARNING: use batchSizeMax
         info,
         &size_internal,
         &size_qr);
    assert(cusolver_status == CUSOLVER_STATUS_SUCCESS);

    printf("numerical factorization needs internal data %lld bytes\n",
 (long long)size_internal);
    printf("numerical factorization needs working space %lld bytes\n",
 (long long)size_qr);

    cudaStat1 = cudaMalloc((void**)&buffer_qr, size_qr);
    assert(cudaStat1 == cudaSuccess);

// 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

```
// step 7: check residual
// xBatch = [x0, x1, x2, ...]
    const int baseA = (CUSPARSE_INDEX_BASE_ONE ==
 cusparseGetMatIndexBase(descrA))? 1:0 ;

    for(int batchId = 0 ; batchId < batchSize; batchId++){
        // measure |bj - Aj*xj|
        double *csrValAj = csrValABatch + batchId * nnzA;
        double *xj = xBatch + batchId * m;
        double *bj = bBatch + batchId * m;
        // sup| bj - Aj*xj|
        double sup_res = 0;
        for(int row = 0 ; row < m ; row++){
            const int start = csrRowPtrA[row ] - baseA;
            const int end   = csrRowPtrA[row+1] - baseA;
            double Ax = 0.0; // Aj(row,:)*xj
            for(int colidx = start ; colidx < end ; colidx++){
                const int col = csrColIndA[colidx] - baseA;
                const double Areg = csrValAj[colidx];
                const double xreg = xj[col];
                Ax = Ax + Areg * xreg;
            }
            double r = bj[row] - Ax;
            sup_res = (sup_res > fabs(r))? sup_res : fabs(r);
        }
        printf("batchId %d: sup|bj - Aj*xj| = %E \n", batchId, sup_res);
    }

    for(int batchId = 0 ; batchId < batchSize; batchId++){
        double *xj = xBatch + batchId * m;
        for(int row = 0 ; row < m ; row++){
            printf("x%d[%d] = %E\n", batchId, row, xj[row]);
        }
        printf("\n");
    }

    return 0;
}
```

Appendix C. QR 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 gerf.

Step 2: B := Q^T*B by ormqr.

Step 3: solve R*X = B by trsm.

The usual includes and main definition

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include ormqr_example.cpp
 *   nvcc -o a.out ormqr_example.o -L/usr/local/cuda/lib64 -lcublas -lcusolver
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <cuda_runtime.h>

#include <cublas_v2.h>
#include <cudense.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[])
{
    cudsHandle_t cudenseH = 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 |
 *   A = | 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;

    int info_gpu = 0;

    const double one = 1;

    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 cudense/cublas handle
    cusolver_status = cudsCreate(&cudenseH);
    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_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 = cudsDgeqrf_bufferSize(
        cudenseH,
        m,
        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 = cudsDgeqrf(
        cudenseH,
        m,
        m,
        d_A,
        lda,
        d_tau,
        d_work,
        lwork,
        devInfo);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
    assert(cudaSuccess == cudaStat1);

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

    printf("after geqrf: info_gpu = %d\n", info_gpu);
    assert(0 == info_gpu);

// step 5: compute Q^T*B
    cusolver_status= cudsDormqr(
        cudenseH,
        CUBLAS_SIDE_LEFT,
        CUBLAS_OP_T,
        m,
        nrhs,
        m,
        d_A,
        lda,
        d_tau,
        d_B,
        ldb,
        d_work,
        lwork,
        devInfo);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSOLVER_STATUS_SUCCESS == cusolver_status);
    assert(cudaSuccess == cudaStat1);
```

Check the results

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

    printf("after ormqr: info_gpu = %d\n", info_gpu);
    assert(0 == info_gpu);

// step 6: compute x = R \ Q^T*B

    cublas_status = cublasDtrsm(
         cublasH,
         CUBLAS_SIDE_LEFT,
         CUBLAS_FILL_MODE_UPPER,
         CUBLAS_OP_N,
         CUBLAS_DIAG_NON_UNIT,
         m,
         nrhs,
         &one,
         d_A,
         lda,
         d_B,
         ldb);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUBLAS_STATUS_SUCCESS == cublas_status);
    assert(cudaSuccess == cudaStat1);

    cudaStat1 = cudaMemcpy(XC, d_B, sizeof(double)*ldb*nrhs,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);

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

// free resources
    if (d_A    ) cudaFree(d_A);
    if (d_tau  ) cudaFree(d_tau);
    if (d_B    ) cudaFree(d_B);
    if (devInfo) cudaFree(devInfo);
    if (d_work ) cudaFree(d_work);


    if (cublasH ) cublasDestroy(cublasH);
    if (cudenseH) cudsDestroy(cudenseH);

    cudaDeviceReset();

    return 0;
}
```

Appendix D. ACKNOWLEDGEMENTS

NVIDIA would like to thank the following individuals and institutions for their contributions:

- CPU LAPACK routines from netlib, LAPACK 3.5.0 (http://www.netlib.org/lapack/)

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.

Appendix E. BIBLIOGRAPHY

[1] Timothy A. Davis, Direct Methods for sparse Linear Systems, siam 2006.

[2] E. Chuthill and J. McKee, reducing the bandwidth of sparse symmetric matrices, ACM '69 Proceedings of the 1969 24th national conference, Pages 157-172.

[3] Alan George, Joseph W. H. Liu, An Implementation of a Pseudoperipheral Node Finder, ACM Transactions on Mathematical Software (TOMS) Volume 5 Issue 3, Sept. 1979 Pages 284-295.

[4] J. R. Gilbert and T. Peierls, Sparse partial pivoting in time proportional to arithmetic operations, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862-874.

[5] Alan George and Esmond Ng, An Implementation of Gaussian Elimination with Partial Pivoting for Sparse Systems, SIAM J. Sci. and Stat. Comput., 6(2), 390-409.

[6] Alan George and Esmond Ng, Symbolic Factorization for Sparse Gaussian Elimination with Paritial Pivoting, SIAM J. Sci. and Stat. Comput., 8(6), 877-898.