CUSPARSE LIBRARY

DU-06709-001_v10.1 | April 2019

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on top of the NVIDIA® CUDA™ runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++. The library routines can be classified into four categories:

- Level 1: operations between a vector in sparse format and a vector in dense format
- Level 2: operations between a matrix in sparse format and a vector in dense format
- Level 3: operations between a matrix in sparse format and a set of vectors in dense format (which can also usually be viewed as a dense tall matrix)
- Conversion: operations that allow conversion between different matrix formats, and compression of csr matrices.

The cuSPARSE library allows developers to access the computational resources of the NVIDIA graphics processing unit (GPU), although it does not auto-parallelize across multiple GPUs. The cuSPARSE API assumes that input and output data reside in GPU (device) memory, unless it is explicitly indicated otherwise by the string **`DevHostPtr`** in a function parameter's name (for example, the parameter **`*resultDevHostPtr`** in the function **`cusparse<t>doti()`**).

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 cuSPARSE library requires hardware with 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. Naming Conventions

The cuSPARSE library functions are available for data types **`float`**, **`double`**, **`cuComplex`**, and **`cuDoubleComplex`**. The sparse Level 1, Level 2, and Level 3 functions follow this naming convention:

cusparse<**t**>[<**matrix data format**>]<**operation**>[<**output matrix data format**>]

where <**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**> can be **dense**, **coo**, **csr**, **csc**, or **hyb**, corresponding to the dense, coordinate, compressed sparse row, compressed sparse column, and hybrid storage formats, respectively.

Finally, the <**operation**> can be **axpyi**, **doti**, **dotci**, **gthr**, **gthrz**, **roti**, or **sctr**, corresponding to the Level 1 functions; it also can be **mv** or **sv**, corresponding to the Level 2 functions, as well as **mm** or **sm**, corresponding to the Level 3 functions.

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

1.2. Asynchronous Execution

The cuSPARSE library functions are executed asynchronously with respect to the host and may return control to the application on the host before the result is ready. Developers can use the **cudaDeviceSynchronize()** function to ensure that the execution of a particular cuSPARSE library routine has completed.

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

1.3. Static Library support

Starting with release 6.5, the cuSPARSE Library is also delivered in a static form as libcusparse_static.a on Linux and Mac OSes. The static cuSPARSE library and all others static maths libraries depend on a common thread abstraction layer library called libculibos.a on Linux and Mac and culibos.lib on Windows.

For example, on linux, to compile a small application using cuSPARSE against the dynamic library, the following command can be used:

```
nvcc myCusparseApp.c  -lcusparse  -o myCusparseApp
```

Whereas to compile against the static cuSPARSE library, the following command has to be used:

```
nvcc myCusparseApp.c  -lcusparse_static   -lculibos -o myCusparseApp
```

It is also possible to use the native Host C++ compiler. Depending on the Host Operating system, some additional libraries like **pthread** or **dl** might be needed on the linking line. The following command on Linux is suggested :

```
    g++ myCusparseApp.c  -lcusparse_static   -lculibos -lcudart_static -
lpthread -ldl -I <cuda-toolkit-path>/include -L <cuda-toolkit-path>/lib64 -o
 myCusparseApp
```

Note that in the latter case, the library **cuda** is not needed. The CUDA Runtime will try to open explicitly the **cuda** library if needed. In the case of a system which does not have the CUDA driver installed, this allows the application to gracefully manage this issue and potentially run if a CPU-only path is available.

Chapter 2. USING THE CUSPARSE API

This chapter describes how to use the cuSPARSE library API. It is not a reference for the cuSPARSE 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. However, simultaneous read/writes of the same objects (or of the same handle) are not safe. Hence the handle must be private per thread, i.e., only one handle per thread is safe.

2.2. Scalar Parameters

In the cuSPARSE API, the scalar parameters α and β can be passed by reference on the host or the device.

The few functions that return a scalar result, such as **`doti()`** and **`nnz()`**, return the resulting value by reference on the host or the device. Even though these functions return immediately, similarly to those that return matrix and vector results, the scalar result is not ready until execution of the routine on the GPU completes. This requires proper synchronization be used when reading the result from the host.

This feature allows the cuSPARSE library functions to execute completely asynchronously using streams, even when α and β are generated by a previous kernel. This situation arises, for example, when the library is used to implement iterative methods for the solution of linear systems and eigenvalue problems [3].

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 cuSPARSE library routine by calling **`cusparseSetStream()`** just before calling the actual cuSPARSE 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.

When streams are used, we recommend using the new cuSPARSE API with scalar parameters and results passed by reference in the device memory to achieve maximum computational overlap.

Although a developer can create many streams, in practice it is not possible to have more than 16 concurrent kernels executing at the same time.

Chapter 3. CUSPARSE INDEXING AND DATA FORMATS

The cuSPARSE library supports dense and sparse vector, and dense and sparse matrix formats.

3.1. Index Base Format

The library supports zero- and one-based indexing. The index base is selected through the **`cusparseIndexBase_t`** type, which is passed as a standalone parameter or as a field in the matrix descriptor **`cusparseMatDescr_t`** type.

3.2. Vector Formats

This section describes dense and sparse vector formats.

3.2.1. Dense Format

Dense vectors are represented with a single data array that is stored linearly in memory, such as the following 7×1 dense vector.

$$[1.0 \quad 0.0 \quad 0.0 \quad 2.0 \quad 3.0 \quad 0.0 \quad 4.0]$$

(This vector is referenced again in the next section.)

3.2.2. Sparse Format

Sparse vectors are represented with two arrays.

- The *data array* has the nonzero values from the equivalent array in dense format.
- The *integer index array* has the positions of the corresponding nonzero values in the equivalent array in dense format.

For example, the dense vector in section 3.2.1 can be stored as a sparse vector with one-based indexing.

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1 & 4 & 5 & 7 \end{bmatrix}$$

It can also be stored as a sparse vector with zero-based indexing.

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 0 & 3 & 4 & 6 \end{bmatrix}$$

In each example, the top row is the data array and the bottom row is the index array, and it is assumed that the indices are provided in increasing order and that each index appears only once.

3.3. Matrix Formats

Dense and several sparse formats for matrices are discussed in this section.

3.3.1. Dense Format

The dense matrix **x** is assumed to be stored in column-major format in memory and is represented by the following parameters.

m	(integer)	The number of rows in the matrix.
n	(integer)	The number of columns in the matrix.
ldx	(integer)	The leading dimension of **x**, which must be greater than or equal to **m**. If **ldx** is greater than **m**, then **x** represents a sub-matrix of a larger matrix stored in memory
x	(pointer)	Points to the data array containing the matrix elements. It is assumed that enough storage is allocated for **x** to hold all of the matrix elements and that cuSPARSE library functions may access values outside of the sub-matrix, but will never overwrite them.

For example, **m×n** dense matrix **x** with leading dimension **ldx** can be stored with one-based indexing as shown.

$$\begin{bmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,n} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m,1} & X_{m,2} & \cdots & X_{m,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{ldX,1} & X_{ldX,2} & \cdots & X_{ldX,n} \end{bmatrix}$$

Its elements are arranged linearly in memory in the order below.

$$[X_{1,1} \;\; X_{2,1} \;\; \cdots \;\; X_{m,1} \;\; \cdots \;\; X_{ldX,1} \;\; \cdots \;\; X_{1,n} \;\; X_{2,n} \;\; \cdots \;\; X_{m,n} \;\; \cdots \;\; X_{ldX,n}]$$

This format and notation are similar to those used in the NVIDIA CUDA cuBLAS library.

3.3.2. Coordinate Format (COO)

The **m×n** sparse matrix **A** is represented in COO format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
cooValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of **A** in row-major format.
cooRowIndA	(pointer)	Points to the integer array of length **nnz** that contains the row indices of the corresponding elements in array **cooValA**.
cooColIndA	(pointer)	Points to the integer array of length **nnz** that contains the column indices of the corresponding elements in array **cooValA**.

A sparse matrix in COO format is assumed to be stored in row-major format: the index arrays are first sorted by row indices and then within the same row by compressed column indices. It is assumed that each pair of row and column indices appears only once.

For example, consider the following 4×5 matrix **A**.

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

It is stored in COO format with zero-based indexing this way.

$$\begin{aligned} \text{cooValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{cooRowIndA} &= [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3 \quad] \\ \text{cooColIndA} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 3 \quad 4 \quad 2 \quad 4 \quad] \end{aligned}$$

In the COO format with one-based indexing, it is stored as shown.

$$\begin{aligned} \text{cooValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{cooRowIndA} &= [1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4 \quad 4 \quad] \\ \text{cooColIndA} &= [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 4 \quad 5 \quad 3 \quad 5 \quad] \end{aligned}$$

3.3.3. Compressed Sparse Row Format (CSR)

The only way the CSR differs from the COO format is that the array containing the row indices is compressed in CSR format. The **m×n** sparse matrix **A** is represented in CSR format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
csrValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of A in row-major format.
csrRowPtrA	(pointer)	Points to the integer array of length **m+1** that holds indices into the arrays **csrColIndA** and **csrValA**. The first **m** entries of this array contain the indices of the first nonzero element in the **i**th row for **i=i,...,m**, while the last entry contains **nnz+csrRowPtrA(0)**. In general, **csrRowPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.

csrColIndA	(pointer)	Points to the integer array of length **nnz** that contains the column indices of the corresponding elements in array **csrValA**.

Sparse matrices in CSR format are assumed to be stored in row-major CSR format, in other words, the index arrays are first sorted by row indices and then within the same row by column indices. It is assumed that each pair of row and column indices appears only once.

Consider again the 4×5 matrix **A**.

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

It is stored in CSR format with zero-based indexing as shown.

$$\begin{aligned} \text{csrValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{csrRowPtrA} &= [0 \quad 2 \quad 4 \quad 7 \quad 9 \quad] \\ \text{csrColIndA} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 3 \quad 4 \quad 2 \quad 4 \quad] \end{aligned}$$

This is how it is stored in CSR format with one-based indexing.

$$\begin{aligned} \text{csrValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{csrRowPtrA} &= [1 \quad 3 \quad 5 \quad 8 \quad 10 \quad] \\ \text{csrColIndA} &= [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 4 \quad 5 \quad 3 \quad 5 \quad] \end{aligned}$$

3.3.4. Compressed Sparse Column Format (CSC)

The CSC format is different from the COO format in two ways: the matrix is stored in column-major format, and the array containing the column indices is compressed in CSC format. The **m×n** matrix **A** is represented in CSC format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
cscValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of **A** in column-major format.
cscRowIndA	(pointer)	Points to the integer array of length **nnz** that contains the row indices of the corresponding elements in array **cscValA**.
cscColPtrA	(pointer)	Points to the integer array of length **n+1** that holds indices into the arrays **cscRowIndA** and **cscValA**. The first **n** entries of this array contain the indices of the first nonzero element in the **i**th row for **i=i,...,n**, while the last entry contains **nnz+cscColPtrA(0)**. In general, **cscColPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.

The matrix **A** in CSR format has exactly the same memory layout as its transpose in CSC format (and vice versa).

For example, consider once again the 4×5 matrix **A**.

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

It is stored in CSC format with zero-based indexing this way.

$$\begin{aligned} \text{cscValA} &= [1.0 \quad 5.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 9.0 \quad 7.0 \quad 8.0 \quad 6.0] \\ \text{cscRowIndA} &= [0 \quad 2 \quad 0 \quad 1 \quad 1 \quad 3 \quad 2 \quad 2 \quad 3 \ ] \\ \text{cscColPtrA} &= [0 \quad 2 \quad 4 \quad 6 \quad 7 \quad 9 \ ] \end{aligned}$$

In CSC format with one-based indexing, this is how it is stored.

$$\begin{aligned} \text{cscValA} &= [1.0 \quad 5.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 9.0 \quad 7.0 \quad 8.0 \quad 6.0] \\ \text{cscRowIndA} &= [1 \quad 3 \quad 1 \quad 2 \quad 2 \quad 4 \quad 3 \quad 3 \quad 4 \ ] \\ \text{cscColPtrA} &= [1 \quad 3 \quad 5 \quad 7 \quad 8 \quad 10 \ ] \end{aligned}$$

Each pair of row and column indices appears only once.

3.3.5. Ellpack-Itpack Format (ELL)

An **m×n** sparse matrix **A** with at most **k** nonzero elements per row is stored in the Ellpack-Itpack (ELL) format [2] using two dense arrays of dimension **m×k**. The first data array contains the values of the nonzero elements in the matrix, while the second integer array contains the corresponding column indices.

For example, consider the 4×5 matrix **A**.

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

This is how it is stored in ELL format with zero-based indexing.

$$\text{data} = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 2.0 & 3.0 & 0.0 \\ 5.0 & 7.0 & 8.0 \\ 9.0 & 6.0 & 0.0 \end{bmatrix}$$

$$\text{indices} = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 2 & -1 \\ 0 & 3 & 4 \\ 2 & 4 & -1 \end{bmatrix}$$

It is stored this way in ELL format with one-based indexing.

$$\text{data} = \begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 2.0 & 3.0 & 0.0 \\ 5.0 & 7.0 & 8.0 \\ 9.0 & 6.0 & 0.0 \end{bmatrix}$$

$$\text{indices} = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 3 & -1 \\ 1 & 4 & 5 \\ 3 & 5 & -1 \end{bmatrix}$$

Sparse matrices in ELL format are assumed to be stored in column-major format in memory. Also, rows with less than **k** nonzero elements are padded in the **data** and **indices** arrays with zero and -1, respectively.

The ELL format is not supported directly, but it is used to store the regular part of the matrix in the HYB format that is described in the next section.

3.3.6. Hybrid Format (HYB)

The HYB sparse storage format is composed of a regular part, usually stored in ELL format, and an irregular part, usually stored in COO format [1]. The ELL and COO parts are always stored using zero-based indexing. HYB is implemented as an opaque data format that requires the use of a conversion operation to store a matrix in it. The conversion operation partitions the general matrix into the regular and irregular parts automatically or according to developer-specified criteria.

For more information, please refer to the description of **cusparseHybPartition_t** type, as well as the description of the conversion routines **dense2hyb**, **csc2hyb** and **csr2hyb**.

3.3.7. Block Compressed Sparse Row Format (BSR)

The only difference between the CSR and BSR formats is the format of the storage element. The former stores primitive data types (**single**, **double**, **cuComplex**, and **cuDoubleComplex**) whereas the latter stores a two-dimensional square block of primitive data types. The dimension of the square block is $blockDim$. The **m×n** sparse matrix **A** is equivalent to a block sparse matrix A_b with $mb = \frac{m + blockDim - 1}{blockDim}$ block rows and $nb = \frac{n + blockDim - 1}{blockDim}$ block columns. If m or n is not multiple of $blockDim$, then zeros are filled into A_b.

A is represented in BSR format by the following parameters.

blockDim	(integer)	Block dimension of matrix **A**.
mb	(integer)	The number of block rows of **A**.
nb	(integer)	The number of block columns of **A**.
nnzb	(integer)	The number of nonzero blocks in the matrix.
bsrValA	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all elements of nonzero blocks of **A**. The block elements are stored in either column-major order or row-major order.
bsrRowPtrA	(pointer)	Points to the integer array of length **mb+1** that holds indices into the arrays **bsrColIndA** and **bsrValA**. The first **mb** entries of this array contain the indices of the first nonzero block in the **i**th block row for **i=1,...,mb**, while the last entry contains **nnzb+bsrRowPtrA(0)**. In general, **bsrRowPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.
bsrColIndA	(pointer)	Points to the integer array of length **nnzb** that contains the column indices of the corresponding blocks in array **bsrValA**.

As with CSR format, (row, column) indices of BSR are stored in row-major order. The index arrays are first sorted by row indices and then within the same row by column indices.

For example, consider again the **4×5** matrix **A**.

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

If *blockDim* is equal to 2, then *mb* is 2, *nb* is 3, and matrix **A** is split into **2×3** block matrix A_b. The dimension of A_b is **4×6**, slightly bigger than matrix A, so zeros are filled in the last column of A_b. The element-wise view of A_b is this.

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

Based on zero-based indexing, the block-wise view of A_b can be represented as follows.

$$A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

The basic element of BSR is a nonzero A_{ij} block, one that contains at least one nonzero element of **A**. Five of six blocks are nonzero in A_b.

$$A_{00} = \begin{bmatrix} 1 & 4 \\ 0 & 2 \end{bmatrix}, A_{01} = \begin{bmatrix} 0 & 0 \\ 3 & 0 \end{bmatrix}, A_{10} = \begin{bmatrix} 5 & 0 \\ 0 & 0 \end{bmatrix}, A_{11} = \begin{bmatrix} 0 & 7 \\ 9 & 0 \end{bmatrix}, A_{12} = \begin{bmatrix} 8 & 0 \\ 6 & 0 \end{bmatrix}$$

BSR format only stores the information of nonzero blocks, including block indices (*i*, *j*) and values A_{ij}. Also row indices are compressed in CSR format.

$$\begin{aligned} \text{bsrValA} &= [A_{00}\ A_{01}\ A_{10}\ A_{11}\ A_{12}] \\ \text{bsrRowPtrA} &= [0\quad 2\quad 5] \\ \text{bsrColIndA} &= [0\quad 1\quad 0\quad 1\quad 2] \end{aligned}$$

There are two ways to arrange the data element of block A_{ij}: row-major order and column-major order. Under column-major order, the physical storage of **bsrValA** is this.

$$bsrValA = [1\ 0\ 4\ 2\ |\ 0\ 3\ 0\ 0\ |\ 5\ 0\ 0\ 0\ |\ 0\ 9\ 7\ 0\ |\ 8\ 6\ 0\ 0\ ]$$

Under row-major order, the physical storage of **bsrValA** is this.

$$bsrValA = [1\ 4\ 0\ 2\ |\ 0\ 0\ 3\ 0\ |\ 5\ 0\ 0\ 0\ |\ 0\ 7\ 9\ 0\ |\ 8\ 0\ 6\ 0\ ]$$

Similarly, in BSR format with one-based indexing and column-major order, **A** can be represented by the following.

$$A_b = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$bsrValA = [1\ 0\ 4\ 2\ |\ 0\ 3\ 0\ 0\ |\ 5\ 0\ 0\ 0\ |\ 0\ 9\ 7\ 0\ |\ 8\ 6\ 0\ 0\ ]$$

$$\text{bsrRowPtrA} = [1 \quad 3 \quad 6]$$
$$\text{bsrColIndA} = [1 \quad 2 \quad 1 \quad 2 \quad 3]$$

The general BSR format has two parameters, **rowBlockDim** and **colBlockDim**. **rowBlockDim** is number of rows within a block and **colBlockDim** is number of columns within a block. If **rowBlockDim=colBlockDim**, general BSR format is the same as BSR format. If **rowBlockDim=colBlockDim=1**, general BSR format is the same as CSR format. The conversion routine gebsr2gebsr is used to do conversion among CSR, BSR and general BSR.

In the cuSPARSE Library, the storage format of blocks in BSR format can be column-major or row-major, independently of the base index. However, if the developer uses BSR format from the Math Kernel Library (MKL) and wants to directly interface with the cuSPARSE Library, then **cusparseDirection_t CUSPARSE_DIRECTION_COLUMN** should be used if the base index is one; otherwise, **cusparseDirection_t CUSPARSE_DIRECTION_ROW** should be used.

3.3.8. Extended BSR Format (BSRX)

BSRX is the same as the BSR format, but the array **bsrRowPtrA** is separated into two parts. The first nonzero block of each row is still specified by the array **bsrRowPtrA**, which is the same as in BSR, but the position next to the last nonzero block of each row is specified by the array **bsrEndPtrA**. Briefly, BSRX format is simply like a 4-vector variant of BSR format.

Matrix **A** is represented in BSRX format by the following parameters.

blockDim	(integer)	Block dimension of matrix **A**.
mb	(integer)	The number of block rows of **A**.
nb	(integer)	The number of block columns of **A**.
nnzb	(integer)	number of nonzero blocks in the matrix **A**.
bsrValA	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all the elements of the nonzero blocks of **A**. The block elements are stored in either column-major order or row-major order.
bsrRowPtrA	(pointer)	Points to the integer array of length **mb** that holds indices into the arrays **bsrColIndA** and **bsrValA**; **bsrRowPtrA(i)** is the position of the first nonzero block of the **i**th block row in **bsrColIndA** and **bsrValA**.
bsrEndPtrA	(pointer)	Points to the integer array of length **mb** that holds indices into the arrays **bsrColIndA** and **bsrValA**; **bsrRowPtrA(i)** is the position next to the last nonzero block of the **i**th block row in **bsrColIndA** and **bsrValA**.
bsrColIndA	(pointer)	Points to the integer array of length **nnzb** that contains the column indices of the corresponding blocks in array **bsrValA**.

A simple conversion between BSR and BSRX can be done as follows. Suppose the developer has a **2×3** block sparse matrix A_b represented as shown.

$$A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

Assume it has this BSR format.

$$\begin{aligned} \text{bsrValA of BSR} &= [A_{00}\ A_{01}\ A_{10}\ A_{11}\ A_{12}] \\ \text{bsrRowPtrA of BSR} &= [0\quad 2\quad 5] \\ \text{bsrColIndA of BSR} &= [0\quad 1\quad 0\quad 1\quad 2] \end{aligned}$$

The **bsrRowPtrA** of the BSRX format is simply the first two elements of the **bsrRowPtrA** BSR format. The **bsrEndPtrA** of BSRX format is the last two elements of the **bsrRowPtrA** of BSR format.

$$\begin{aligned} \text{bsrRowPtrA of BSRX} &= [0\quad 2] \\ \text{bsrEndPtrA of BSRX} &= [2\quad 5] \end{aligned}$$

The advantage of the BSRX format is that the developer can specify a submatrix in the original BSR format by modifying **bsrRowPtrA** and **bsrEndPtrA** while keeping **bsrColIndA** and **bsrValA** unchanged.

For example, to create another block matrix $\tilde{A} = \begin{bmatrix} O & O & O \\ O & A_{11} & O \end{bmatrix}$ that is slightly different from A, the developer can keep **bsrColIndA** and **bsrValA**, but reconstruct $\tilde{A}$ by properly setting of **bsrRowPtrA** and **bsrEndPtrA**. The following 4-vector characterizes $\tilde{A}$.

$$\begin{aligned} \text{bsrValA of } \tilde{A} &= [A_{00}\ A_{01}\ A_{10}\ A_{11}\ A_{12}] \\ \text{bsrColIndA of } \tilde{A} &= [0\quad 1\quad 0\quad 1\quad 2] \\ \text{bsrRowPtrA of } \tilde{A} &= [0\quad 3] \\ \text{bsrEndPtrA of } \tilde{A} &= [0\quad 4] \end{aligned}$$

Chapter 4. CUSPARSE TYPES REFERENCE

4.1. Data 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**.

4.2. cusparseAction_t

This type indicates whether the operation is performed only on indices or on data and indices.

Value	Meaning
CUSPARSE_ACTION_SYMBOLIC	the operation is performed only on indices.
CUSPARSE_ACTION_NUMERIC	the operation is performed on data and indices.

4.3. cusparseDirection_t

This type indicates whether the elements of a dense matrix should be parsed by rows or by columns (assuming column-major storage in memory of the dense matrix) in function cusparse[S|D|C|Z]nnz. Besides storage format of blocks in BSR format is also controlled by this type.

Value	Meaning
CUSPARSE_DIRECTION_ROW	the matrix should be parsed by rows.
CUSPARSE_DIRECTION_COLUMN	the matrix should be parsed by columns.

4.4. cusparseHandle_t

This is a pointer type to an opaque cuSPARSE context, which the user must initialize by calling prior to calling **cusparseCreate()** any other library function. The handle created and returned by **cusparseCreate()** must be passed to every cuSPARSE function.

4.5. cusparseHybMat_t

This is a pointer type to an opaque structure holding the matrix in HYB format, which is created by **cusparseCreateHybMat** and destroyed by **cusparseDestroyHybMat**.

4.5.1. cusparseHybPartition_t

This type indicates how to perform the partitioning of the matrix into regular (ELL) and irregular (COO) parts of the HYB format.

The partitioning is performed during the conversion of the matrix from a dense or sparse format into the HYB format and is governed by the following rules. When **CUSPARSE_HYB_PARTITION_AUTO** is selected, the cuSPARSE library automatically decides how much data to put into the regular and irregular parts of the HYB format. When **CUSPARSE_HYB_PARTITION_USER** is selected, the width of the regular part of the HYB format should be specified by the caller. When **CUSPARSE_HYB_PARTITION_MAX** is selected, the width of the regular part of the HYB format equals to the maximum number of non-zero elements per row, in other words, the entire matrix is stored in the regular part of the HYB format.

The *default* is to let the library automatically decide how to split the data.

Value	Meaning
CUSPARSE_HYB_PARTITION_AUTO	the automatic partitioning is selected (*default*).
CUSPARSE_HYB_PARTITION_USER	the user specified treshold is used.
CUSPARSE_HYB_PARTITION_MAX	the data is stored in ELL format.

4.6. cusparseMatDescr_t

This structure is used to describe the shape and properties of a matrix.

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

4.6.1. cusparseDiagType_t

This type indicates if the matrix diagonal entries are unity. The diagonal elements are always assumed to be present, but if **CUSPARSE_DIAG_TYPE_UNIT** is passed to an API routine, then the routine assumes that all diagonal entries are unity and will not read or modify those entries. Note that in this case the routine assumes the diagonal entries are equal to one, regardless of what those entries are actually set to in memory.

Value	Meaning
CUSPARSE_DIAG_TYPE_NON_UNIT	the matrix diagonal has non-unit elements.
CUSPARSE_DIAG_TYPE_UNIT	the matrix diagonal has unit elements.

4.6.2. cusparseFillMode_t

This type indicates if the lower or upper part of a matrix is stored in sparse storage.

Value	Meaning
CUSPARSE_FILL_MODE_LOWER	the lower triangular part is stored.
CUSPARSE_FILL_MODE_UPPER	the upper triangular part is stored.

4.6.3. cusparseIndexBase_t

This type indicates if the base of the matrix indices is zero or one.

Value	Meaning
CUSPARSE_INDEX_BASE_ZERO	the base index is zero.
CUSPARSE_INDEX_BASE_ONE	the base index is one.

4.6.4. cusparseMatrixType_t

This type indicates the type of matrix stored in sparse storage. Notice that for symmetric, Hermitian and triangular matrices only their lower or upper part is assumed to be stored.

The whole idea of matrix type and fill mode is to keep minimum storage for symmetric/Hermitian matrix, and also to take advantage of symmetric property on SpMV (Sparse Matrix Vector multiplication). To compute **y=A*x** when **A** is symmetric and only lower triangular part is stored, two steps are needed. First step is to compute **y=(L+D)*x** and second step is to compute **y=L^T*x + y**. Given the fact that the transpose operation **y=L^T*x** is 10x slower than non-transpose version **y=L*x**, the symmetric property does not show up any performance gain. It is better for the user to extend the symmetric matrix to a general matrix and apply **y=A*x** with matrix type **CUSPARSE_MATRIX_TYPE_GENERAL**.

In general, SpMV, preconditioners (incomplete Cholesky or incomplete LU) and triangular solver are combined together in iterative solvers, for example PCG and

GMRES. If the user always uses general matrix (instead of symmetric matrix), there is no need to support other than general matrix in preconditioners. Therefore the new routines, **[bsr|csr]sv2** (triangular solver), **[bsr|csr]ilu02** (incomplete LU) and **[bsr|csr]ic02** (incomplete Cholesky), only support matrix type **CUSPARSE_MATRIX_TYPE_GENERAL**.

Value	Meaning
CUSPARSE_MATRIX_TYPE_GENERAL	the matrix is general.
CUSPARSE_MATRIX_TYPE_SYMMETRIC	the matrix is symmetric.
CUSPARSE_MATRIX_TYPE_HERMITIAN	the matrix is Hermitian.
CUSPARSE_MATRIX_TYPE_TRIANGULAR	the matrix is triangular.

4.7. cusparseOperation_t

This type indicates which operations need to be performed with the sparse matrix.

Value	Meaning
CUSPARSE_OPERATION_NON_TRANSPOSE	the non-transpose operation is selected.
CUSPARSE_OPERATION_TRANSPOSE	the transpose operation is selected.
CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE	the conjugate transpose operation is selected.

4.8. cusparsePointerMode_t

This type indicates whether the scalar values are passed by reference on the host or device. It is important to point out that if several scalar values are passed by reference in the function call, all of them will conform to the same single pointer mode. The pointer mode can be set and retrieved using **cusparseSetPointerMode()** and **cusparseGetPointerMode()** routines, respectively.

Value	Meaning
CUSPARSE_POINTER_MODE_HOST	the scalars are passed by reference on the host.
CUSPARSE_POINTER_MODE_DEVICE	the scalars are passed by reference on the device.

4.9. cusparseAlgMode_t

This is type for algorithm parameter to cusparseCsrmvEx() and cusparseCsrmvEx_bufferSize() functions. Note that previously defined values **CUSPARSE_ALG0** (for naive algorithm) and **CUSPARSE_ALG1** (for merge path algorithm) are deprecated and replaced by named aliases specified below.

Value	Meaning
CUSPARSE_ALG_NAIVE	Use default naive algorithm.
CUSPARSE_ALG_MERGE_PATH	Use load-balancing algorithm that suits better for irregular nonzero-patterns.

4.10. cusparseSolvePolicy_t

This type indicates whether level information is generated and used in **csrsv2, csric02, csrilu02, bsrsv2, bsric02 and bsrilu02**.

Value	Meaning
CUSPARSE_SOLVE_POLICY_NO_LEVEL	no level information is generated and used.
CUSPARSE_SOLVE_POLICY_USE_LEVEL	generate and use level information.

4.11. cusparseSolveAnalysisInfo_t

This is a pointer type to an opaque structure holding the information collected in the analysis phase of the solution of the sparse triangular linear system. It is expected to be passed unchanged to the solution phase of the sparse triangular linear system.

4.12. cusparseSolveAnalysisInfo_t

This is a pointer type to an opaque structure holding the information collected in the analysis phase of the solution of the sparse triangular linear system. It is expected to be passed unchanged to the solution phase of the sparse triangular linear system.

4.13. csrsv2Info_t

This is a pointer type to an opaque structure holding the information used in **csrsv2_bufferSize()**, **csrsv2_analysis()**, and **csrsv2_solve()**.

4.14. csrsm2Info_t

This is a pointer type to an opaque structure holding the information used in **csrsm2_bufferSize()**, **csrsm2_analysis()**, and **csrsm2_solve()**.

4.15. csric02Info_t

This is a pointer type to an opaque structure holding the information used in **csric02_bufferSize()**, **csric02_analysis()**, and **csric02()**.

4.16. csrilu02Info_t

This is a pointer type to an opaque structure holding the information used in **csrilu02_bufferSize()**, **csrilu02_analysis()**, and **csrilu02()**.

4.17. bsrsv2Info_t

This is a pointer type to an opaque structure holding the information used in **bsrsv2_bufferSize()**, **bsrsv2_analysis()**, and **bsrsv2_solve()**.

4.18. bsrsm2Info_t

This is a pointer type to an opaque structure holding the information used in **bsrsm2_bufferSize()**, **bsrsm2_analysis()**, and **bsrsm2_solve()**.

4.19. bsric02Info_t

This is a pointer type to an opaque structure holding the information used in **bsric02_bufferSize()**, **bsric02_analysis()**, and **bsric02()**.

4.20. bsrilu02Info_t

This is a pointer type to an opaque structure holding the information used in **bsrilu02_bufferSize()**, **bsrilu02_analysis()**, and **bsrilu02()**.

4.21. csrgemm2Info_t

This is a pointer type to an opaque structure holding the information used in **csrgemm2_bufferSizeExt()**, and **csrgemm2()**.

4.22. cusparseStatus_t

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

CUSPARSE_STATUS_SUCCESS	The operation completed successfully.
CUSPARSE_STATUS_NOT_INIT	The cuSPARSE 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 cuSPARSE routine, or an error in the hardware setup.

	To correct: call **cusparseCreate()** prior to the function call; and check that the hardware, an appropriate version of the driver, and the cuSPARSE library are correctly installed.
CUSPARSE_STATUS_ALLOC_FA	Resource allocation failed inside the cuSPARSE library. This is usually caused by a **cudaMalloc()** failure. **To correct:** prior to the function call, deallocate previously allocated memory as much as possible.
CUSPARSE_STATUS_INVALID_	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.
CUSPARSE_STATUS_ARCH_MIS	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 appropriate compute capability, which is 1.1 for 32-bit atomic operations and 1.3 for double precision.
CUSPARSE_STATUS_MAPPING_	An access to GPU memory space failed, which is usually caused by a failure to bind a texture. **To correct:** prior to the function call, unbind any previously bound textures.
CUSPARSE_STATUS_EXECUTIO	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 cuSPARSE library are correctly installed.
CUSPARSE_STATUS_INTERNAL_	An internal cuSPARSE 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 cuSPARSE 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.
CUSPARSE_STATUS_MATRIX_T	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 **cusparseMatDescr_t descrA** were set correctly.

Chapter 5. CUSPARSE HELPER FUNCTION REFERENCE

The cuSPARSE helper functions are described in this section.

5.1. cusparseCreate()

```
cusparseStatus_t
cusparseCreate(cusparseHandle_t *handle)
```

This function initializes the cuSPARSE library and creates a handle on the cuSPARSE context. It must be called before any other cuSPARSE API function is invoked. It allocates hardware resources necessary for accessing the GPU.

Output

handle	the pointer to the handle to the cuSPARSE context.

Status Returned

CUSPARSE_STATUS_SUCCESS	the initialization succeeded.
CUSPARSE_STATUS_NOT_INITIALIZED	the CUDA Runtime initialization failed.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_ARCH_MISMATCH	the device compute capability (CC) is less than 1.1. The CC of at least 1.1 is required.

5.2. cusparseCreateSolveAnalysisInfo()

```
cusparseStatus_t
cusparseCreateSolveAnalysisInfo(cusparseSolveAnalysisInfo_t *info)
```

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

Input

info	the pointer to the solve and analysis structure.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.3. cusparseCreateHybMat()

```
cusparseStatus_t
cusparseCreateHybMat(cusparseHybMat_t *hybA)
```

This function creates and initializes the **hybA** opaque data structure.

Input

hybA	the pointer to the hybrid format storage structure.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.4. cusparseCreateMatDescr()

```
cusparseStatus_t
cusparseCreateMatDescr(cusparseMatDescr_t *descrA)
```

This function initializes the matrix descriptor. It sets the fields **MatrixType** and **IndexBase** to the *default* values **CUSPARSE_MATRIX_TYPE_GENERAL** and **CUSPARSE_INDEX_BASE_ZERO** , respectively, while leaving other fields uninitialized.

Input

descrA	the pointer to the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the descriptor was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.5. cusparseCreateSolveAnalysisInfo()

```
cusparseStatus_t
cusparseCreateSolveAnalysisInfo(cusparseSolveAnalysisInfo_t *info)
```

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

Input

info	the pointer to the solve and analysis structure.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.

CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.6. cusparseDestroy()

```
cusparseStatus_t
cusparseDestroy(cusparseHandle_t handle)
```

This function releases CPU-side resources used by the cuSPARSE library. The release of GPU-side resources may be deferred until the application shuts down.

Input

handle	the handle to the cuSPARSE context.

Status Returned

CUSPARSE_STATUS_SUCCESS	the shutdown succeeded.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

5.7. cusparseDestroySolveAnalysisInfo()

```
cusparseStatus_t
cusparseDestroySolveAnalysisInfo(cusparseSolveAnalysisInfo_t info)
```

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

Input

info	the solve and analysis structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.8. cusparseDestroyHybMat()

```
cusparseStatus_t
cusparseDestroyHybMat(cusparseHybMat_t hybA)
```

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

Input

hybA	the hybrid format storage structure.

Status Returned

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.9. cusparseDestroyMatDescr()

```
cusparseStatus_t
cusparseDestroyMatDescr(cusparseMatDescr_t descrA)
```

This function releases the memory allocated for the matrix descriptor.

Input

descrA	the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.10. cusparseDestroySolveAnalysisInfo()

```
cusparseStatus_t
cusparseDestroySolveAnalysisInfo(cusparseSolveAnalysisInfo_t info)
```

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

Input

info	the solve and analysis structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.11. cusparseGetLevelInfo()

```
cusparseStatus_t
cusparseGetLevelInfo(cusparseHandle_t handle,
                     cusparseSolveAnalysisInfo_t info,
                     int *nlevels,
                     int **levelPtr,
                     int **levelInd)
```

This function returns the number of levels and the assignment of rows into the levels computed by either the csrsv_analysis, csrsm_analysis or hybsv_analysis routines.

Input

handle	handle to the cuSPARSE library context.
info	the pointer to the solve and analysis structure.

Output

nlevels	number of levels.

levelPtr	integer array of **nlevels+1** elements that contains the start of every level and the end of the last level plus one.
levelInd	integer array of **m** (number of rows in the matrix) elements that contains the row indices belonging to every level.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library or the solve analysis structure was not initialized.

5.12. cusparseGetMatDiagType()

```
cusparseDiagType_t
cusparseGetMatDiagType(const cusparseMatDescr_t descrA)
```

This function returns the **DiagType** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated diagType types.

5.13. cusparseGetMatFillMode()

```
cusparseFillMode_t
cusparseGetMatFillMode(const cusparseMatDescr_t descrA)
```

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

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated fillMode types.

5.14. cusparseGetMatIndexBase()

```
cusparseIndexBase_t
cusparseGetMatIndexBase(const cusparseMatDescr_t descrA)
```

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

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated indexBase types.

5.15. cusparseGetMatType()

```
cusparseMatrixType_t
cusparseGetMatType(const cusparseMatDescr_t descrA)
```

This function returns the **MatrixType** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated matrix types.

5.16. cusparseGetPointerMode()

```
cusparseStatus_t
cusparseGetPointerMode(cusparseHandlet handle,
                       cusparsePointerMode_t *mode)
```

This function obtains the pointer mode used by the cuSPARSE library. Please see the section on the **cusparsePointerMode_t** type for more details.

Input

handle	the handle to the cuSPARSE context.

Output

mode	One of the enumerated pointer mode types.

Status Returned

CUSPARSE_STATUS_SUCCESS	the pointer mode was returned successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

5.17. cusparseGetVersion()

```
cusparseStatus_t
cusparseGetVersion(cusparseHandle_t handle, int *version)
```

This function returns the version number of the cuSPARSE library.

Input

handle	the handle to the cuSPARSE context.

Output

version	the version number of the library.

Status Returned

CUSPARSE_STATUS_SUCCESS	the version was returned successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

5.18. cusparseSetMatDiagType()

```
cusparseStatus_t
cusparseSetMatDiagType(cusparseMatDescr_t descrA,
                       cusparseDiagType_t diagType)
```

This function sets the **DiagType** field of the matrix descriptor **descrA**.

Input

diagType	One of the enumerated diagType types.

Output

descrA	the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the field **DiagType** was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	An invalid **diagType** parameter was passed.

5.19. cusparseSetMatFillMode()

```
cusparseStatus_t
cusparseSetMatFillMode(cusparseMatDescr_t descrA,
                       cusparseFillMode_t fillMode)
```

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

Input

fillMode	One of the enumerated fillMode types.

Output

descrA	the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the **FillMode** field was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	An invalid **fillMode** parameter was passed.

5.20. cusparseSetMatIndexBase()

```
cusparseStatus_t
cusparseSetMatIndexBase(cusparseMatDescr_t descrA,
                        cusparseIndexBase_t base)
```

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

Input

base	One of the enumerated indexBase types.

Output

descrA	the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the **IndexBase** field was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	An invalid **base** parameter was passed.

5.21. cusparseSetMatType()

```
cusparseStatus_t
cusparseSetMatType(cusparseMatDescr_t descrA, cusparseMatrixType_t type)
```

This function sets the **MatrixType** field of the matrix descriptor **descrA**.

Input

type	One of the enumerated matrix types.

Output

descrA	the matrix descriptor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the **MatrixType** field was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	An invalid **type** parameter was passed.

5.22. cusparseSetPointerMode()

```
cusparseStatus_t
cusparseSetPointerMode(cusparseHandle_t handle,
                       cusparsePointerMode_t mode)
```

This function sets the pointer mode used by the cuSPARSE library. The *default* is for the values to be passed by reference on the host. Please see the section on the **cublasPointerMode_t** type for more details.

Input

handle	the handle to the cuSPARSE context.
mode	One of the enumerated pointer mode types.

Status Returned

CUSPARSE_STATUS_SUCCESS	the pointer mode was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	the library was not initialized.

5.23. cusparseSetStream()

```
cusparseStatus_t
cusparseSetStream(cusparseHandle_t handle, cudaStream_t streamId)
```

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

Input

handle	the handle to the cuSPARSE context.
streamId	the stream to be used by the library.

Status Returned

CUSPARSE_STATUS_SUCCESS	the stream was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	the library was not initialized.

5.24. cusparseGetStream()

```
cusparseStatus_t
cusparseGetStream(cusparseHandle_t handle, cudaStream_t *streamId)
```

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

Input

handle	the handle to the cuSPARSE context.

Output

streamId	the stream to be used by the library.

Status Returned

CUSPARSE_STATUS_SUCCESS	the stream was returned successfully.
CUSPARSE_STATUS_INVALID_VALUE	the library was not initialized.

5.25. cusparseCreateCsrsv2Info()

```
cusparseStatus_t
cusparseCreateCsrsv2Info(csrsv2Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of csrsv2.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.26. cusparseDestroyCsrsv2Info()

```
cusparseStatus_t
cusparseDestroyCsrsv2Info(csrsv2Info_t info);
```

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

Input

info	the solve **(csrsv2_solve)** and analysis **(csrsv2_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.27. cusparseCreateCsrsm2Info()

```
cusparseStatus_t
cusparseCreateCsrsm2Info(csrsm2Info_t *info);
```

This function creates and initializes the solve and analysis structure of csrsm2 to *default* values.

Input

info	the pointer to the solve and analysis structure of csrsm2.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.28. cusparseDestroyCsrsm2Info()

```
cusparseStatus_t
cusparseDestroyCsrsm2Info(csrsm2Info_t info);
```

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

Input

info	the solve **(csrsm2_solve)** and analysis **(csrsm2_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.29. cusparseCreateCsric02Info()

```
cusparseStatus_t
cusparseCreateCsric02Info(csric02Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of incomplete Cholesky.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.30. cusparseDestroyCsric02Info()

```
cusparseStatus_t
cusparseDestroyCsric02Info(csric02Info_t info);
```

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

Input

info	the solve **(csric02_solve)** and analysis **(csric02_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.31. cusparseCreateCsrilu02Info()

```
cusparseStatus_t
cusparseCreateCsrilu02Info(csrilu02Info_t *info);
```

This function creates and initializes the solve and analysis structure of incomplete LU to *default* values.

Input

info	the pointer to the solve and analysis structure of incomplete LU.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.32. cusparseDestroyCsrilu02Info()

```
cusparseStatus_t
cusparseDestroyCsrilu02Info(csrilu02Info_t info);
```

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

Input

info	the solve **(csrilu02_solve)** and analysis **(csrilu02_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.33. cusparseCreateBsrsv2Info()

```
cusparseStatus_t
cusparseCreateBsrsv2Info(bsrsv2Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of bsrsv2.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.34. cusparseDestroyBsrsv2Info()

```
cusparseStatus_t
cusparseDestroyBsrsv2Info(bsrsv2Info_t info);
```

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

Input

info	the solve **(bsrsv2_solve)** and analysis **(bsrsv2_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.35. cusparseCreateBsrsm2Info()

```
cusparseStatus_t
cusparseCreateBsrsm2Info(bsrsm2Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of bsrsm2.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.36. cusparseDestroyBsrsm2Info()

```
cusparseStatus_t
cusparseDestroyBsrsm2Info(bsrsm2Info_t info);
```

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

Input

info	the solve **(bsrsm2_solve)** and analysis **(bsrsm2_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.37. cusparseCreateBsric02Info()

```
cusparseStatus_t
cusparseCreateBsric02Info(bsric02Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of block incomplete Cholesky.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.38. cusparseDestroyBsric02Info()

```
cusparseStatus_t
cusparseDestroyBsric02Info(bsric02Info_t info);
```

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

Input

info	the solve **(bsric02_solve)** and analysis **(bsric02_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.39. cusparseCreateBsrilu02Info()

```
cusparseStatus_t
cusparseCreateBsrilu02Info(bsrilu02Info_t *info);
```

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

Input

info	the pointer to the solve and analysis structure of block incomplete LU.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.40. cusparseDestroyBsrilu02Info()

```
cusparseStatus_t
cusparseDestroyBsrilu02Info(bsrilu02Info_t info);
```

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

Input

info	the solve **(bsrilu02_solve)** and analysis **(bsrilu02_analysis)** structure.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.41. cusparseCreateCsrgemm2Info()

```
cusparseStatus_t
cusparseCreateCsrgemm2Info(csrgemm2Info_t *info);
```

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

Input

info	the pointer to the analysis structure of general sparse matrix-matrix multiplication.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.42. cusparseDestroyCsrgemm2Info()

```
cusparseStatus_t
cusparseDestroyCsrgemm2Info(csrgemm2Info_t info);
```

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

Input

info	opaque structure of csrgemm2.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.43. cusparseCreatePruneInfo()

```
cusparseStatus_t
cusparseCreatePruneInfo(pruneInfo_t *info);
```

This function creates and initializes structure of **prune** to *default* values.

Input

info	the pointer to the structure of **prune**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the structure was initialized successfully.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

5.44. cusparseDestroyPruneInfo()

```
cusparseStatus_t
cusparseDestroyPruneInfo(pruneInfo_t info);
```

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

Input

info	the structure of **prune**.

Status Returened

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

Chapter 6. CUSPARSE LEVEL 1 FUNCTION REFERENCE

This chapter describes sparse linear algebra functions that perform operations between dense and sparse vectors.

6.1. cusparse<t>axpyi()

```
cusparseStatus_t
cusparseSaxpyi(cusparseHandle_t handle, int nnz,
               const float           *alpha,
               const float           *xVal, const int *xInd,
               float           *y, cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDaxpyi(cusparseHandle_t handle, int nnz,
               const double          *alpha,
               const double          *xVal, const int *xInd,
               double          *y, cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCaxpyi(cusparseHandle_t handle, int nnz,
               const cuComplex       *alpha,
               const cuComplex       *xVal, const int *xInd,
               cuComplex       *y, cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZaxpyi(cusparseHandle_t handle, int nnz,
               const cuDoubleComplex *alpha,
               const cuDoubleComplex *xVal, const int *xInd,
               cuDoubleComplex *y, cusparseIndexBase_t idxBase)
```

This function multiplies the vector **x** in sparse format by the constant α and adds the result to the vector **y** in dense format. This operation can be written as

$$y = y + \alpha * x$$

In other words,

```
for i=0 to nnz-1
    y[xInd[i]-idxBase] = y[xInd[i]-idxBase] + alpha*xVal[i]
```

This function requires no extra storage. It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
alpha	<type> scalar used for multiplication.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

y	<type> updated vector in dense format (that is unchanged if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

6.2. cusparse<t>doti()

```
cusparseStatus_t
cusparseSdoti(cusparseHandle_t handle, int nnz,
              const float           *xVal,
              const int *xInd, const float           *y,
              float           *resultDevHostPtr,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDdoti(cusparseHandle_t handle, int nnz,
              const double          *xVal,
              const int *xInd, const double          *y,
              double          *resultDevHostPtr,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCdoti(cusparseHandle_t handle, int nnz,
              const cuComplex       *xVal,
              const int *xInd, const cuComplex       *y,
              cuComplex       *resultDevHostPtr,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZdoti(cusparseHandle_t handle, int nnz, const
              cuDoubleComplex *xVal,
              const int *xInd, const cuDoubleComplex *y,
              cuDoubleComplex *resultDevHostPtr,
              cusparseIndexBase_t idxBase)
```

This function returns the dot product of a vector **x** in sparse format and vector **y** in dense format. This operation can be written as

$$result = y^T x$$

In other words,

```
for i=0 to nnz-1
    resultDevHostPtr += xVal[i]*y[xInd[i-idxBase]]
```

This function requires some temporary extra storage that is allocated internally. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
resultDevHostPtr	pointer to the location of the result in the device or host memory.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

resultDevHostPtr	scalar result in the device or host memory (that is zero if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the reduction buffer could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

6.3. cusparse<t>dotci()

```
cusparseStatus_t
cusparseCdotci(cusparseHandle_t handle, int nnz,
               const cuComplex       *xVal,
               const int *xInd, const cuComplex       *y,
               cuComplex       *resultDevHostPtr, cusparseIndexBase_t
 idxBase)
cusparseStatus_t
cusparseZdotci(cusparseHandle_t handle, int nnz,
               const cuDoubleComplex *xVal,
               const int *xInd, const cuDoubleComplex *y,
               cuDoubleComplex *resultDevHostPtr, cusparseIndexBase_t
 idxBase)
```

This function returns the dot product of a complex conjugate of vector **x** in sparse format and vector **y** in dense format. This operation can be written as

$$result = x^H y$$

In other words,

```
for i=0 to nnz-1
 resultDevHostPtr += conj(xVal[i])*y[xInd[i-idxBase]]
```

This function requires some temporary extra storage that is allocated internally. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
resultDevHostPtr	pointer to the location of the result in the device or host memory.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

resultDevHostPtr	scalar result in the device or host memory (that is zero if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the reduction buffer could not be allocated.

CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

6.4. cusparse<t>gthr()

```
cusparseStatus_t
cusparseSgthr(cusparseHandle_t handle, int nnz,
              const float           *y,
              float           *xVal, const int *xInd,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDgthr(cusparseHandle_t handle, int nnz,
              const double          *y,
              double          *xVal, const int *xInd,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCgthr(cusparseHandle_t handle, int nnz,
              const cuComplex       *y,
              cuComplex       *xVal, const int *xInd,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZgthr(cusparseHandle_t handle, int nnz,
              const cuDoubleComplex *y,
              cuDoubleComplex *xVal, const int *xInd,
              cusparseIndexBase_t idxBase)
```

This function gathers the elements of the vector **y** listed in the index array **xInd** into the data array **xVal**.

This function requires no extra storage. It is executed asynchronously with respect to the host and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
y	<type> vector in dense format (of **size≥max(xInd)-idxBase+1**).
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> vector with **nnz** nonzero values that were gathered from vector **y** (that is unchanged if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

6.5. cusparse<t>gthrz()

```
cusparseStatus_t
cusparseSgthrz(cusparseHandle_t handle, int nnz, float           *y,
               float           *xVal, const int *xInd,
               cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDgthrz(cusparseHandle_t handle, int nnz, double          *y,
               double          *xVal, const int *xInd,
               cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCgthrz(cusparseHandle_t handle, int nnz, cuComplex       *y,
               cuComplex       *xVal, const int *xInd,
               cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZgthrz(cusparseHandle_t handle, int nnz, cuDoubleComplex *y,
               cuDoubleComplex *xVal, const int *xInd,
               cusparseIndexBase_t idxBase)
```

This function gathers the elements of the vector **y** listed in the index array **xInd** into the data array **xVal**. Also, it zeros out the gathered elements in the vector **y**.

This function requires no extra storage. It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
y	<type> vector in dense format (of **size≥max(xInd)-idxBase+1**).
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> vector with **nnz** nonzero values that were gathered from vector **y** (that is unchanged if **nnz == 0**).

y	<type> vector in dense format with elements indexed by **xInd** set to zero (it is unchanged if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

6.6. cusparse<t>roti()

```
cusparseStatus_t
cusparseSroti(cusparseHandle_t handle, int nnz, float  *xVal,
              const int *xInd,
              float  *y, const float  *c, const float  *s,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDroti(cusparseHandle_t handle, int nnz, double *xVal,
              const int *xInd,
              double *y, const double *c, const double *s,
              cusparseIndexBase_t idxBase)
```

This function applies the Givens rotation matrix

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

to sparse **x** and dense **y** vectors. In other words,

```
for i=0 to nnz-1
    y[xInd[i]-idxBase] = c * y[xInd[i]-idxBase] - s*xVal[i]
    x[i]               = c * xVal[i]            + s * y[xInd[i]-idxBase]
```

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
c	cosine element of the rotation matrix.
s	sine element of the rotation matrix.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> updated vector in sparse format (that is unchanged if **nnz == 0**).
y	<type> updated vector in dense format (that is unchanged if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

6.7. cusparse<t>sctr()

```
cusparseStatus_t
cusparseSsctr(cusparseHandle_t handle, int nnz,
              const float           *xVal,
              const int *xInd, float           *y,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDsctr(cusparseHandle_t handle, int nnz,
              const double          *xVal,
              const int *xInd, double          *y,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCsctr(cusparseHandle_t handle, int nnz,
              const cuComplex       *xVal,
              const int *xInd, cuComplex       *y,
              cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZsctr(cusparseHandle_t handle, int nnz,
              const cuDoubleComplex *xVal,
              const int *xInd, cuDoubleComplex *y,
              cusparseIndexBase_t idxBase)
```

This function scatters the elements of the vector **x** in sparse format into the vector **y** in dense format. It modifies only the elements of **y** whose indices are listed in the array **xInd**.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
xVal	<type> vector with **nnz** nonzero values of vector **x**.

xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> dense vector (of **size≥max(xInd)–idxBase+1**).
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

y	<type> vector with **nnz** nonzero values that were scattered from vector **x** (that is unchanged if **nnz == 0**).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the **idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**..
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

Chapter 7. CUSPARSE LEVEL 2 FUNCTION REFERENCE

This chapter describes the sparse linear algebra functions that perform operations between sparse matrices and dense vectors.

In particular, the solution of sparse triangular linear systems is implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate **`csrsv_analysis()`** function. The analysis is specific to the sparsity pattern of the given matrix and to the selected **`cusparseOperation_t`** type. The information from the analysis phase is stored in the parameter of type **`cusparseSolveAnalysisInfo_t`** that has been initialized previously with a call to **`cusparseCreateSolveAnalysisInfo()`**.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the **`cusparseSolveAnalysisInfo_t`** parameter by calling the appropriate **`csrsv_solve()`** function. The solve phase may be performed multiple times with different right-hand sides, while the analysis phase needs to be performed only once. This is especially useful when a sparse triangular linear system must be solved for a set of different right-hand sides one at a time, while its coefficient matrix remains the same.

Finally, once all the solves have completed, the opaque data structure pointed to by the **`cusparseSolveAnalysisInfo_t`** parameter can be released by calling **`cusparseDestroySolveAnalysisInfo()`**. For more information please refer to [3].

7.1. cusparse<t>bsrmv()

```
cusparseStatus_t
cusparseSbsrmv(cusparseHandle_t handle, cusparseDirection_t dir,
    cusparseOperation_t trans, int mb, int nb, int nnzb,
    const float *alpha, const cusparseMatDescr_t descr,
    const float *bsrVal, const int *bsrRowPtr, const int *bsrColInd,
    int  blockDim, const float *x,
    const float *beta, float *y)
cusparseStatus_t
cusparseDbsrmv(cusparseHandle_t handle, cusparseDirection_t dir,
    cusparseOperation_t trans, int mb, int nb, int nnzb,
    const double *alpha, const cusparseMatDescr_t descr,
    const double *bsrVal, const int *bsrRowPtr, const int *bsrColInd,
    int  blockDim, const double *x,
    const double *beta, double *y)
cusparseStatus_t
cusparseCbsrmv(cusparseHandle_t handle, cusparseDirection_t dir,
    cusparseOperation_t trans, int mb, int nb, int nnzb,
    const cuComplex *alpha, const cusparseMatDescr_t descr,
    const cuComplex *bsrVal, const int *bsrRowPtr, const int *bsrColInd,
    int  blockDim, const cuComplex *x,
    const cuComplex *beta, cuComplex *y)
cusparseStatus_t
cusparseZbsrmv(cusparseHandle_t handle, cusparseDirection_t dir,
    cusparseOperation_t trans, int mb, int nb, int nnzb,
    const cuDoubleComplex *alpha, const cusparseMatDescr_t descr,
    const cuDoubleComplex *bsrVal, const int *bsrRowPtr, const int
 *bsrColInd,
    int  blockDim, const cuDoubleComplex *x,
    const cuDoubleComplex *beta, cuDoubleComplex *y)
```

This function performs the matrix-vector operation

$$y = \alpha * op(A) * x + \beta * y$$

where A is an $(mb * blockDim) \times (nb * blockDim)$ sparse matrix that is defined in BSR storage format by the three arrays **bsrVal**, **bsrRowPtr**, and **bsrColInd**); **x** and **y** are vectors; α and β are scalars; and

$$op(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Several comments on **bsrmv()**:

- Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported, that is

$$y = \alpha * A * x + \beta * y$$

- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported.
- The size of vector **x** should be $(nb * blockDim)$ at least, and the size of vector **y** should be $(mb * blockDim)$ at least; otherwise, the kernel may return **CUSPARSE_STATUS_EXECUTION_FAILED** because of an out-of-bounds array.

For example, suppose the user has a CSR format and wants to try **bsrmv()**, the following code demonstrates how to use **csr2bsr()** conversion and **bsrmv()** multiplication in single precision.

```
// Suppose that A is m x n sparse matrix represented by CSR format,
// hx is a host vector of size n, and hy is also a host vector of size m.
// m and n are not multiple of blockDim.
// step 1: transform CSR to BSR with column-major order
int base, nnz;
int nnzb;
cusparseDirection_t dirA = CUSPARSE_DIRECTION_COLUMN;
int mb = (m + blockDim-1)/blockDim;
int nb = (n + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
cusparseXcsr2bsrNnz(handle, dirA, m, n,
        descrA, csrRowPtrA, csrColIndA, blockDim,
        descrC, bsrRowPtrC, &nnzb);
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dirA, m, n,
        descrA, csrValA, csrRowPtrA, csrColIndA, blockDim,
        descrC, bsrValC, bsrRowPtrC, bsrColIndC);
// step 2: allocate vector x and vector y large enough for bsrmv
cudaMalloc((void**)&x, sizeof(float)*(nb*blockDim));
cudaMalloc((void**)&y, sizeof(float)*(mb*blockDim));
cudaMemcpy(x, hx, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(y, hy, sizeof(float)*m, cudaMemcpyHostToDevice);
// step 3: perform bsrmv
cusparseSbsrmv(handle, dirA, transA, mb, nb, nnzb, &alpha,
   descrC, bsrValC, bsrRowPtrC, bsrColIndC, blockDim, x, &beta, y);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
trans	the operation $\mathrm{op}(A)$. Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported.
mb	number of block rows of matrix A.
nb	number of block columns of matrix A.
nnzb	number of nonzero blocks of matrix A.
alpha	<type> scalar used for multiplication.
descr	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**.
bsrVal	<type> array of **nnz** ($=$ **csrRowPtrA(mb)** $-$ **csrRowPtrA(0)**) nonzero blocks of matrix A.
bsrRowPtr	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.

bsrColInd	integer array of **nnz** (= **csrRowPtrA(mb) - csrRowPtrA(0)**) column indices of the nonzero blocks of matrix A.
blockDim	block dimension of sparse matrix A, larger than zero.
x	<type> vector of $nb * blockDim$ elements.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of $mb * blockDim$ elements.

Output

y	<type> updated vector.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**, **trans != CUSPARSE_OPERATION_NON_TRANSPOSE**, $blockDim < 1$, **dir** is not row-major or column-major, or **IndexBase of descr** is not base-0 or base-1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.2. cusparse<t>bsrxmv()

```
cusparseStatus_t
cusparseSbsrxmv(cusparseHandle_t handle,
                cusparseDirection_t dir,
                cusparseOperation_t trans,
                int sizeOfMask,
                int mb,
                int nb,
                int nnzb,
                const float *alpha,
                const cusparseMatDescr_t descr,
                const float *bsrVal,
                const int *bsrMaskPtr,
                const int *bsrRowPtr,
                const int *bsrEndPtr,
                const int *bsrColInd,
                int blockDim,
                const float *x,
                const float *beta,
                float *y)
cusparseStatus_t
cusparseDbsrxmv(cusparseHandle_t handle,
                cusparseDirection_t dir,
                cusparseOperation_t trans,
                int sizeOfMask,
                int mb,
                int nb,
                int nnzb,
                const double *alpha,
                const cusparseMatDescr_t descr,
                const double *bsrVal,
                const int *bsrMaskPtr,
                const int *bsrRowPtr,
                const int *bsrEndPtr,
                const int *bsrColInd,
                int blockDim,
                const double *x,
                const double *beta,
                double *y)
cusparseStatus_t
cusparseCbsrxmv(cusparseHandle_t handle,
                cusparseDirection_t dir,
                cusparseOperation_t trans,
                int sizeOfMask,
                int mb,
                int nb,
                int nnzb,
                const cuComplex *alpha,
                const cusparseMatDescr_t descr,
                const cuComplex *bsrVal,
                const int *bsrMaskPtr,
                const int *bsrRowPtr,
                const int *bsrEndPtr,
                const int *bsrColInd,
                int blockDim,
                const cuComplex *x,
                const cuComplex *beta,
                cuComplex *y)
cusparseStatus_t
cusparseZbsrxmv(cusparseHandle_t handle,
                cusparseDirection_t dir,
                cusparseOperation_t trans,
                int sizeOfMask,
                int mb,
```

This function performs a **bsrmv** and a mask operation

$$y(mask) = (\alpha * op(A) * x + \beta * y)(mask)$$

where A is an $(mb * blockDim) \times (nb * blockDim)$ sparse matrix that is defined in BSRX storage format by the four arrays **bsrVal**, **bsrRowPtr**, **bsrEndPtr**, and **bsrColInd**); **x** and **y** are vectors; α and β are scalars; and

$$op(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The mask operation is defined by array **bsrMaskPtr** which contains updated block row indices of y. If row i is not specified in **bsrMaskPtr**, then **bsrxmv()** does not touch row block i of A and y.

For example, consider the 2×3 block matrix A:

$$A = \begin{bmatrix} A_{11} & A_{12} & O \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

and its one-based BSR format (three vector form) is

$$\begin{aligned} \text{bsrVal} &= [A_{11} \;\; A_{12} \;\; A_{21} \;\; A_{22} \;\; A_{23}] \\ \text{bsrRowPtr} &= [1 \;\; 3 \;\; 6] \\ \text{bsrColInd} &= [1 \;\; 2 \;\; 1 \;\; 2 \;\; 3] \end{aligned}$$

Suppose we want to do the following **bsrmv** operation on a matrix $\overline{A}$ which is slightly different from A.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} := alpha * \left(\tilde{A} = \begin{bmatrix} O & O & O \\ O & A_{22} & O \end{bmatrix} \right) * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} y_1 \\ beta * y_2 \end{bmatrix}$$

We don't need to create another BSR format for the new matrix $\overline{A}$, all that we should do is to keep **bsrVal** and **bsrColInd** unchanged, but modify **bsrRowPtr** and add an additional array **bsrEndPtr** which points to the last nonzero elements per row of $\overline{A}$ plus 1.

For example, the following **bsrRowPtr** and **bsrEndPtr** can represent matrix $\overline{A}$:

$$\begin{aligned} \text{bsrRowPtr} &= [1 \;\; 4] \\ \text{bsrEndPtr} &= [1 \;\; 5] \end{aligned}$$

Further we can use a mask operator (specified by array **bsrMaskPtr**) to update particular block row indices of y only because y_1 is never changed. In this case, **bsrMaskPtr** = [2] and **sizeOfMask**=1.

The mask operator is equivalent to the following operation:

$$\begin{bmatrix} ? \\ y_2 \end{bmatrix} := alpha * \begin{bmatrix} ? & ? & ? \\ O & A_{22} & O \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + beta * \begin{bmatrix} ? \\ y_2 \end{bmatrix}$$

If a block row is not present in the **bsrMaskPtr**, then no calculation is performed on that row, and the corresponding value in **y** is unmodified. The question mark "?" is used to inidcate row blocks not in **bsrMaskPtr**.

In this case, first row block is not present in **bsrMaskPtr**, so **bsrRowPtr[0]** and **bsrEndPtr[0]** are not touched also.

```
bsrRowPtr = [?    4]
bsrEndPtr = [?    5]
```

A couple of comments on **bsrxmv()**:

- Only **CUSPARSE_OPERATION_NON_TRANSPOSE** and **CUSPARSE_MATRIX_TYPE_GENERAL** are supported.
- Parameters **bsrMaskPtr**, **bsrRowPtr**, **bsrEndPtr** and **bsrColInd** are consistent with base index, either one-based or zero-based. The above example is one-based.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
trans	the operation $op(A)$. Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported.
sizeOfMask	number of updated block rows of y.
mb	number of block rows of matrix A.
nb	number of block columns of matrix A.
nnzb	number of nonzero blocks of matrix A.
alpha	<type> scalar used for multiplication.
descr	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**.
bsrVal	<type> array of **nnz** nonzero blocks of matrix A.
bsrMaskPtr	integer array of **sizeOfMask** elements that contains the indices corresponding to updated block rows.
bsrRowPtr	integer array of **mb** elements that contains the start of every block row.
bsrEndPtr	integer array of **mb** elements that contains the end of the every block row plus one.
bsrColInd	integer array of **nnzb** column indices of the nonzero blocks of matrix A.
blockDim	block dimension of sparse matrix A, larger than zero.

x	<type> vector of $nb * blockDim$ elements.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of $mb * blockDim$ elements.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**, **trans != CUSPARSE_OPERATION_NON_TRANSPOSE**, $blockDim < 1$, **dir** is not row-major or column-major, or **IndexBase of descr** is not base-0 or base-1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.3. cusparse<t>csrmv()

```
cusparseStatus_t
cusparseScsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, int nnz, const float           *alpha,
               const cusparseMatDescr_t descrA,
               const float           *csrValA,
               const int *csrRowPtrA, const int *csrColIndA,
               const float           *x, const float           *beta,
               float           *y)
cusparseStatus_t
cusparseDcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, int nnz, const double          *alpha,
               const cusparseMatDescr_t descrA,
               const double          *csrValA,
               const int *csrRowPtrA, const int *csrColIndA,
               const double          *x, const double          *beta,
               double          *y)
cusparseStatus_t
cusparseCcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, int nnz, const cuComplex       *alpha,
               const cusparseMatDescr_t descrA,
               const cuComplex       *csrValA,
               const int *csrRowPtrA, const int *csrColIndA,
               const cuComplex       *x, const cuComplex       *beta,
               cuComplex       *y)
cusparseStatus_t
cusparseZcsrmv(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, int nnz, const cuDoubleComplex *alpha,
               const cusparseMatDescr_t descrA,
               const cuDoubleComplex *csrValA,
               const int *csrRowPtrA, const int *csrColIndA,
               const cuDoubleComplex *x, const cuDoubleComplex *beta,
               cuDoubleComplex *y)
```

This function performs the matrix-vector operation

$$y = \alpha * op(A) * x + \beta * y$$

A is an **m×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are vectors; α and β are scalars; and

$$op(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

When using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this routine may produce slightly different results during different runs with the same input parameters. For these matrix types it uses atomic operations to compute the final result, consequently many threads may be adding floating point numbers to the same memory location without any specific ordering, which may produce slightly different results for each run.

If exactly the same output is required for any input when multiplying by the transpose of a general matrix, the following procedure can be used:

1. Convert the matrix from CSR to CSC format using one of the **csr2csc()** functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.

2. Call the **csrmv()** function with the **cusparseOperation_t** parameter set to **CUSPARSE_OPERATION_NON_TRANSPOSE** and with the interchanged rows and columns of the matrix stored in CSC format. This (implicitly) multiplies the vector by the transpose of the matrix in the original CSR format.

This function requires no extra storage for the general matrices when operation **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It requires some extra storage for Hermitian/symmetric matrices and for the general matrices when an operation different than **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation $\mathrm{op}(A)$.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, **CUSPARSE_MATRIX_TYPE_SYMMETRIC**, and **CUSPARSE_MATRIX_TYPE_HERMITIAN**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
x	<type> vector of **n** elements if $\mathrm{op}(A) = A$, and **m** elements if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of **m** elements if $\mathrm{op}(A) = A$, and **n** elements if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$

Output

y	<type> updated vector.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision (compute capability (c.c.) >= 1.3 required), symmetric/Hermitian matrix (c.c. >= 1.2 required), or transpose operation (c.c. >= 1.1 required).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.4. cusparseCsrmvEx()

```
cusparseStatus_t
cusparseCsrmvEx(cusparseHandle_t handle,
                cusparseAlgMode_t alg,
                cusparseOperation_t transA,
                int m, int n, int nnz,
                const void *alpha, cudaDataType alphatype,
                const cusparseMatDescr_t descrA,
                const void *csrValA, cudaDataType csrValAtype,
                const int *csrRowPtrA,
                const int *csrColIndA,
                const void *x, cudaDataType xtype,
                const void *beta, cudaDataType betatype,
                void *y, cudaDataType ytype,
                cudaDataType executiontype,
                void* buffer);
```

This function is an extended version of **cusparse<t>csrmv()** which performs the matrix-vector multiply operation. For detailed description of the functionality, see **cusparse<t>csrmv()**. Also see **cusparseAlgMode_t** for **alg** parameter description

For alg **CUSPARSE_ALG_NAIVE**: for half-precision execution type, the minimum GPU architecture is SM_53. Also, for both half-precision IO and execution, only CUSPARSE_MATRIX_TYPE_GENERAL and CUSPARSE_OPERATION_NON_TRANSPOSE are supported.

For alg **CUSPARSE_ALG_MERGE_PATH**: half-precision is not supported, only CUSPARSE_MATRIX_TYPE_GENERAL, CUSPARSE_OPERATION_NON_TRANSPOSE and CUSPARSE_INDEX_BASE_ZERO are supported. Input, output and execution types should be the same.

All pointers should be aligned with 128 bytes.

Input specifically required by cusparseCsrmvEx

alg	Algorithm implementation for csrmv, see **cusparseAlgMode_t** for possible values.

alphatype	Data type of **alpha**.
csrValAtype	Data type of **csrValA**.
xtype	Data type of **x**.
betatype	Data type of **beta**.
ytype	Data type of **y**.
executiontype	Data type used for computation.
buffer	Pointer to workspace buffer. Not currently used, however must be allocated and be aligned with word boundaries. Recommended to always use **cusparseCsrmvEx_bufferSize** to obtain the right size for this buffer.

7.5. cusparseCsrmvEx_bufferSize()

```
cusparseCsrmvEx_bufferSize(cusparseHandle_t handle,
                                                cusparseAlgMode_t alg,
                                                cusparseOperation_t
 transA,
                                                int m, int n, int nnz,
                                                const void *alpha,
 cudaDataType alphatype,
                                                const
 cusparseMatDescr_t descrA,
                                                const void *csrValA,
 cudaDataType csrValAtype,
                                                const int *csrRowPtrA,
                                                const int *csrColIndA,
                                                const void *x,
 cudaDataType xtype,
                                                const void *beta,
 cudaDataType betatype,
                                                void *y, cudaDataType
 ytype,
                                                cudaDataType
 executiontype,
                                                size_t
 *bufferSizeInBytes);
```

This function returns the size of the workspace needed by **cusparseCsrmvEx**. User needs to allocate a buffer of this size and give that buffer to **cusparseCsrmvEx** as an argument. All the arguments are similar to **cusparseCsrmvEx** except the following output argument.

Output specifically required by cusparseCsrmvEx_bufferSize

bufferSizeInBytes	Pointer to a **size_t** variable, which will be assigned with the size of workspace needed by **cusparseCsrmv**. Returned number is always more than zero, so **cudaMalloc()** with provided number of bytes needed would not return an error due to number of requested bytes being zero.

7.6. cusparse<t>csrmv_mp()

```
cusparseStatus_t
cusparseScsrmv_mp(cusparseHandle_t handle, cusparseOperation_t transA,
                  int m, int n, int nnz,
                  const float           *alpha,
                  const cusparseMatDescr_t descrA,
                  const float           *csrValA,
                  const int *csrRowPtrA, const int *csrColIndA,
                  const float           *x, const float           *beta,
                  float           *y)
cusparseStatus_t
cusparseDcsrmv_mp(cusparseHandle_t handle, cusparseOperation_t transA,
                  int m, int n, int nnz,
                  const double          *alpha,
                  const cusparseMatDescr_t descrA,
                  const double          *csrValA,
                  const int *csrRowPtrA, const int *csrColIndA,
                  const double          *x, const double          *beta,
                  double          *y)
cusparseStatus_t
cusparseCcsrmv_mp(cusparseHandle_t handle, cusparseOperation_t transA,
                  int m, int n, int nnz,
                  const cuComplex       *alpha,
                  const cusparseMatDescr_t descrA,
                  const cuComplex       *csrValA,
                  const int *csrRowPtrA, const int *csrColIndA,
                  const cuComplex       *x, const cuComplex       *beta,
                  cuComplex       *y)
cusparseStatus_t
cusparseZcsrmv_mp(cusparseHandle_t handle, cusparseOperation_t transA,
                  int m, int n, int nnz,
                  const cuDoubleComplex *alpha,
                  const cusparseMatDescr_t descrA,
                  const cuDoubleComplex *csrValA,
                  const int *csrRowPtrA, const int *csrColIndA,
                  const cuDoubleComplex *x, const cuDoubleComplex *beta,
                  cuDoubleComplex *y)
```

This function performs a load-balanced matrix-vector operation

$$y = \alpha * op(A) * x + \beta * y$$

A is an **m×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are vectors; α and β are scalars; and

$$op(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Note: This function is deprecated in favor of **cusparseCsrmvEx()** with **CUSPARSE_ALG_MERGE_PATH** parameter which provides same functionality with better perfromance.

This routine was introduced specifically to address some of the loss of performance in the regular csrmv() code due to irregular sparsity patterns. The core kernels are based

on the "MergePath" approach created by Duanne Merril. By using this approach, we are able to provide performance independent of a sparsity pattern across data types.

Remark: **csrmv_mp** only supports matrix type **CUSPARSE_MATRIX_TYPE_GENERAL** and **CUSPARSE_OPERATION_NON_TRANSPOSE** operation.

Appendix C provides a simple example of **csrmv_mp**.

Input

handle	handle to the cuSPARSE library context.
trans	the operation $\mathrm{op}(A)$. only support **CUSPARSE_OPERATION_NON_TRANSPOSE**.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
x	<type> vector of **n** elements if $\mathrm{op}(A) = A$, and **m** elements if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of **m** elements if $\mathrm{op}(A) = A$, and **n** elements if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$

Output

y	<type> updated vector.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).

CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision (compute capability (c.c.) >= 1.3 required), symmetric/Hermitian matrix (c.c. >= 1.2 required), or transpose operation (c.c. >= 1.1 required).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.7. cusparse<t>gemvi()

```
cusparseStatus_t
cusparseSgemvi(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, const float           *alpha,
               const float *A,
               int lda, int nnz,
               const float           *x,
               const int             *xInd,
               const float           *beta,
               float                 *y,
               cusparseIndexBase_t    idxBase,
               void                  *pBuffer)
cusparseStatus_t
cusparseDgemvi(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, const double           *alpha,
               const double *A,
               int lda, int nnz,
               const double          *x,
               const int             *xInd,
               const double          *beta,
               double                *y,
               cusparseIndexBase_t    idxBase,
               void                  *pBuffer)
cusparseStatus_t
cusparseCgemvi(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, const cuComplex           *alpha,
               const cuComplex *A,
               int lda, int nnz,
               const cuComplex       *x,
               const int             *xInd,
               const cuComplex       *beta,
               cuComplex             *y,
               cusparseIndexBase_t    idxBase,
               void                  *pBuffer)
cusparseStatus_t
cusparseZgemvi(cusparseHandle_t handle, cusparseOperation_t transA,
               int m, int n, const cuDoubleComplex           *alpha,
               const cuDoubleComplex *A,
               int lda, int nnz,
               const cuDoubleComplex       *x,
               const int             *xInd,
               const cuDoubleComplex       *beta,
               cuDoubleComplex             *y,
               cusparseIndexBase_t    idxBase,
               void                  *pBuffer)
```

This function performs the matrix-vector operation

$$y = \alpha * op(A) * x + \beta * y$$

A is an **m×n** dense matrix and a sparse vector x that is defined in a sparse storage format by the two arrays **xVal**, **xInd** of length **nnz**, and **y** is a dense vector; α and β are scalars; and

$$op(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

To simplify the implementation, we have not (yet) optimized the transpose multiple case. We recommend the following for users interested in this case.

1. Convert the matrix from CSR to CSC format using one of the **csr2csc()** functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.

2. Call the **gemvi()** function with the **cusparseOperation_t** parameter set to **CUSPARSE_OPERATION_NON_TRANSPOSE** and with the interchanged rows and columns of the matrix stored in CSC format. This (implicitly) multiplies the vector by the transpose of the matrix in the original CSR format.

This function requires no extra storage for the general matrices when operation **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It requires some extra storage for Hermitian/symmetric matrices and for the general matrices when an operation different than **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation $op(A)$.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
alpha	<type> scalar used for multiplication.
A	the pointer to dense matrix **A**.
lda	size of the leading dimension of **A**.
nnz	number of nonzero elements of vector **x**.
x	<type> sparse vector of **nnz** elements of size **n** if $op(A) = A$, and size **m** if $op(A) = A^T$ or $op(A) = A^H$
xInd	Indices of non-zero values in **x**
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> dense vector of **m** elements if $op(A) = A$, and **n** elements if $op(A) = A^T$ or $op(A) = A^H$
idxBase	0 or 1, for 0 based or 1 based indexing, respectively

pBuffer	working space buffer, of size given by **Xgemvi_getBufferSize()**

Output

y	<type> updated dense vector.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision (compute capability (c.c.) >= 1.3 required), symmetric/Hermitian matrix (c.c. >= 1.2 required), or transpose operation (c.c. >= 1.1 required).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.8. cusparse<t>gemvi_bufferSize()

```
cusparseStatus_t
cusparseSgemvi_bufferSize(cusparseHandle_t handle,
                          cusparseOperation_t transA,
                          int m,
                          int n,
                          int nnz,
                          int *pBufferSize)

cusparseStatus_t
cusparseDgemvi_bufferSize(cusparseHandle_t handle,
                          cusparseOperation_t transA,
                          int m,
                          int n,
                          int nnz,
                          int *pBufferSize)

cusparseStatus_t
cusparseCgemvi_bufferSize(cusparseHandle_t handle,
                          cusparseOperation_t transA,
                          int m,
                          int n,
                          int nnz,
                          int *pBufferSize)

cusparseStatus_t
cusparseZgemvi_bufferSize(cusparseHandle_t handle,
                          cusparseOperation_t transA,
                          int m,
                          int n,
                          int nnz,
                          int *pBufferSize)
```

This function returns size of buffer used in **gemvi()**

A is an **(m)x(n)** dense matrix.

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Input

handle	handle to the cuSPARSE library context.
transA	the operation **op(A)**.
m	number of rows of matrix **A**.
n	number of columns of matrix **Y**.
nnz	number of nonzero entries of vector **x** multiplying **A**.

Output

pBufferSize	number of elements needed the buffer used in **gemvi()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n, nnz<=0**)
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.9. cusparse<t>bsrsv2_bufferSize()

```
cusparseStatus_t
cusparseSbsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           float *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDbsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           double *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCbsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZbsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrsv2Info_t info,
                           int *pBufferSizeInBytes);
```

This function returns size of the buffer used in **bsrsv2**, a new sparse triangular linear system **op(A)*y = α x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of parameter **trans** and the upper (lower) triangular part of **A**, **bsrsv2_bufferSize()** returns the maximum size buffer among these combinations. The buffer size depends on the dimensions **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsrsv2_bufferSize()** again to have the correct buffer size; otherwise a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\text{op}(A)$.
mb	number of block rows of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A; must be larger than zero.

Output

info	record of internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in the **bsrsv2_analysis()** and **bsrsv2_solve()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.10. cusparse<t>bsrsv2_analysis()

```
cusparseStatus_t
cusparseSbsrsv2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         int mb,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const float *bsrValA,
                         const int *bsrRowPtrA,
                         const int *bsrColIndA,
                         int blockDim,
                         bsrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseDbsrsv2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         int mb,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const double *bsrValA,
                         const int *bsrRowPtrA,
                         const int *bsrColIndA,
                         int blockDim,
                         bsrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseCbsrsv2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         int mb,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuComplex *bsrValA,
                         const int *bsrRowPtrA,
                         const int *bsrColIndA,
                         int blockDim,
                         bsrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseZbsrsv2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         int mb,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex *bsrValA,
                         const int *bsrRowPtrA,
                         const int *bsrColIndA,
                         int blockDim,
                         bsrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);
```

This function performs the analysis phase of **bsrsv2**, a new sparse triangular linear system **op(A)*y** $= \alpha$ **x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The block of BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **bsrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsv2_analysis()** reports a structural zero and computes level information, which stored in the opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **bsrsv2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrsv2_analysis()** always reports the first structural zero, even when parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if block **A(j,j)** is missing for some **j**. The user needs to call **cusparseXbsrsv2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsrsv2_solve()** if **bsrsv2_analysis()** reports a structural zero. In this case, the user can still call **bsrsv2_solve()**, which will return a numerical zero at the same position as a structural zero. However the result **x** is meaningless.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\text{op}(A)$.
mb	number of block rows of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types

	are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.
info	structure initialized using **cusparseCreateBsrsv2Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is return by **bsrsv2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.11. cusparse<t>bsrsv2_solve()

```
cusparseStatus_t
cusparseSbsrsv2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      int mb,
                      int nnzb,
                      const float *alpha,
                      const cusparseMatDescr_t descrA,
                      const float *bsrValA,
                      const int *bsrRowPtrA,
                      const int *bsrColIndA,
                      int blockDim,
                      bsrsv2Info_t info,
                      const float *x,
                      float *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseDbsrsv2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      int mb,
                      int nnzb,
                      const double *alpha,
                      const cusparseMatDescr_t descrA,
                      const double *bsrValA,
                      const int *bsrRowPtrA,
                      const int *bsrColIndA,
                      int blockDim,
                      bsrsv2Info_t info,
                      const double *x,
                      double *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseCbsrsv2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      int mb,
                      int nnzb,
                      const cuComplex *alpha,
                      const cusparseMatDescr_t descrA,
                      const cuComplex *bsrValA,
                      const int *bsrRowPtrA,
                      const int *bsrColIndA,
                      int blockDim,
                      bsrsv2Info_t info,
                      const cuComplex *x,
                      cuComplex *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseZbsrsv2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      int mb,
                      int nnzb,
                      const cuDoubleComplex *alpha,
                      const cusparseMatDescr_t descrA,
                      const cuDoubleComplex *bsrValA,
```

This function performs the solve phase of **bsrsv2**, a new sparse triangular linear system **op(A)*y = α x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrsv02_solve()** can support an arbitrary **blockDim**.

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires a buffer size returned by **bsrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrsv2_solve()** can be done without level information, the user still needs to be aware of consistency. If **bsrsv2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrsv2_solve()** can be run with or without levels. On the other hand, if **bsrsv2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrsv2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance, but may spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **bsrsv2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **bsrsv2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run, and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, and pick the fastest one to perform the remaining iterations.

Function **bsrsv02_solve()** has the same behavior as **csrsv02_solve()**. That is, **bsr2csr(bsrsv02(A)) = csrsv02(bsr2csr(A))**. The numerical zero of **csrsv02_solve()** means there exists some zero **A(j,j)**. The numerical zero of **bsrsv02_solve()** means there exists some block **A(j,j)** that is not invertible.

Function **bsrsv2_solve()** reports the first numerical zero, including a structural zero. No numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is not invertible for some **j**. The user needs to call **cusparseXbsrsv2_zeroPivot()** to know where the numerical zero is.

For example, suppose L is a lower triangular matrix with unit diagonal, then the following code solves **L*y=x** by level information.

```
// Suppose that L is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// L is lower triangular with unit diagonal.
// Assumption:
// - dimension of matrix L is m(=mb*blockDim),
// - matrix L has nnz(=nnzb*blockDim*blockDim) nonzero elements,
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of L on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_x and d_y are of size m.
cusparseMatDescr_t descr = 0;
bsrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal, specified by parameter CUSPARSE_DIAG_TYPE_UNIT
//   (L may not have all diagonal elements.)
cusparseCreateMatDescr(&descr);
cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatFillMode(descr, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr, CUSPARSE_DIAG_TYPE_UNIT);

// step 2: create a empty info structure
cusparseCreateBsrsv2Info(&info);

// step 3: query how much memory used in bsrsv2, and allocate the buffer
cusparseDbsrsv2_bufferSize(handle, dir, trans, mb, nnzb, descr,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &pBufferSize);

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis
cusparseDbsrsv2_analysis(handle, dir, trans, mb, nnzb, descr,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim,
    info, policy, pBuffer);
// L has unit diagonal, so no structural zero is reported.
status = cusparseXbsrsv2_zeroPivot(handle, info, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is missing\n", structural_zero, structural_zero);
}

// step 5: solve L*y = x
cusparseDbsrsv2_solve(handle, dir, trans, mb, nnzb, &alpha, descr,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info,
   d_x, d_y, policy, pBuffer);
// L has unit diagonal, so no numerical zero is reported.
status = cusparseXbsrsv2_zeroPivot(handle, info, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is zero\n", numerical_zero, numerical_zero);
}

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyBsrsv2Info(info);
cusparseDestroyMatDescr(descr);
cusparseDestroy(handle);
```

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\mathrm{op}(A)$.
mb	number of block rows and block columns of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**, larger than zero.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
x	<type> right-hand-side vector of size **m**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **bsrsv2_bufferSize()**.

Output

y	<type> solution vector of size **m**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.

CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.12. cusparseXbsrsv2_zeroPivot()

```
cusparseStatus_t
cusparseXbsrsv2_zeroPivot(cusparseHandle_t handle,
                          bsrsv2Info_t info,
                          int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** is either structural zero or numerical zero (singular block). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsrsv2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains a structural zero or numerical zero if the user already called **bsrsv2_analysis()** or **bsrsv2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

7.13. cusparse<t>csrsv_analysis()

```
cusparseStatus_t
cusparseScsrsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        int m,
                        int nnz,
                        const cusparseMatDescr_t descrA,
                        const float *csrValA,
                        const int *csrRowPtrA,
                        const int *csrColIndA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseDcsrsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        int m,
                        int nnz,
                        const cusparseMatDescr_t descrA,
                        const double *csrValA,
                        const int *csrRowPtrA,
                        const int *csrColIndA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseCcsrsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        int m,
                        int nnz,
                        const cusparseMatDescr_t descrA,
                        const cuComplex *csrValA,
                        const int *csrRowPtrA,
                        const int *csrColIndA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseZcsrsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        int m,
                        int nnz,
                        const cusparseMatDescr_t descrA,
                        const cuDoubleComplex *csrValA,
                        const int *csrRowPtrA,
                        const int *csrColIndA,
                        cusparseSolveAnalysisInfo_t info)
```

This function performs the analysis phase of the solution of a sparse triangular linear system

$$\mathrm{op}(A) * \mathrm{y} = \alpha * \mathrm{x}$$

where **A** is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The routine **csrsv_analysis** supports analysis phase of **csrsv_solve**, **csric0** and **csrilu0**. The user has to be careful of which routine is called after **csrsv_analysis**. The matrix descriptor must be the same for **csrsv_analysis** and its subsequent call to **csrsv_solve**, **csric0** and **csrilu0**.

For **csrsv_solve**, the matrix type must be **CUSPARSE_MATRIX_TYPE_TRIANGULAR** or **CUSPARSE_MATRIX_TYPE_GENERAL**.

For **csrilu0**, the matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**.

For **csric0**, the matrix type must be **CUSPARSE_MATRIX_TYPE_SYMMETRIC** or **CUSPARSE_MATRIX_TYPE_HERMITIAN**.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a significant amount of extra storage that is proportional to the matrix size. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation $\mathrm{op}(A)$
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix A. The supported matrix types are **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and **CUSPARSE_MATRIX_TYPE_GENERAL** for **csrsv_solve**, **CUSPARSE_MATRIX_TYPE_SYMMETRIC** and **CUSPARSE_MATRIX_TYPE_HERMITIAN** for **csric0**, **CUSPARSE_MATRIX_TYPE_GENERAL** for **csrilu0**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure initialized using **cusparseCreateSolveAnalysisInfo**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.14. cusparseCsrsv_analysisEx()

```
cusparseStatus_t cusparseCsrsv_analysisEx(cusparseHandle_t handle,
                                          cusparseOperation_t transA,
                                          int m, int nnz,
                                          const cusparseMatDescr_t
 descrA,
                                          const void *csrSortedValA,
 cudaDataType csrSortedValAtype,
                                          const int *csrSortedRowPtrA,
                                          const int *csrSortedColIndA,
                                          cusparseSolveAnalysisInfo_t
 info,
                                          cudaDataType executiontype);
```

This function is an extended version of **cusparse<t>csrsv_analysis()**. For detailed description of the functionality, see **cusparse<t>csrsv_analysis()**.

This function does not support half-precision execution type, but it supports half-precision IO with single precision execution.

Input specifically required by cusparseCsrsv_analysisEx

csrSortedValAtype	Data type of **csrSortedValA**.
executiontype	Data type used for computation.

7.15. cusparse<t>csrsv_solve()

```
cusparseStatus_t cusparseScsrsv_solve(cusparseHandle_t handle,
                                      cusparseOperation_t transA,
                                      int m, const float *alpha,
                                      const cusparseMatDescr_t descrA,
                                      const float *csrSortedValA,
                                      const int *csrSortedRowPtrA, const
 int *csrSortedColIndA,
                                      cusparseSolveAnalysisInfo_t info,
                                      const float *f, float *x);

cusparseStatus_t cusparseDcsrsv_solve(cusparseHandle_t handle,
                                      cusparseOperation_t transA,
                                      int m, const double *alpha,
                                      const cusparseMatDescr_t descrA,
                                      const double *csrSortedValA,
                                      const int *csrSortedRowPtrA, const
 int *csrSortedColIndA,
                                      cusparseSolveAnalysisInfo_t info,
                                      const double *f, double *x);

cusparseStatus_t cusparseCcsrsv_solve(cusparseHandle_t handle,
                                      cusparseOperation_t transA,
                                      int m, const cuComplex *alpha,
                                      const cusparseMatDescr_t descrA,
                                      const cuComplex *csrSortedValA,
                                      const int *csrSortedRowPtrA, const
 int *csrSortedColIndA,
                                      cusparseSolveAnalysisInfo_t info,
                                      const cuComplex *f, cuComplex *x);

cusparseStatus_t cusparseZcsrsv_solve(cusparseHandle_t handle,
                                      cusparseOperation_t transA,
                                      int m, const cuDoubleComplex
 *alpha,
                                      const cusparseMatDescr_t descrA,
                                      const cuDoubleComplex
 *csrSortedValA,
                                      const int *csrSortedRowPtrA, const
 int *csrSortedColIndA,
                                      cusparseSolveAnalysisInfo_t info,
                                      const cuDoubleComplex *f,
 cuDoubleComplex *x);
```

This function performs the solve phase of the solution of a sparse triangular linear system

$$\mathrm{op}(A) * \mathrm{x} = \alpha * \mathrm{f}$$

where **A** is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrSortedValA**, **csrSortedRowPtrA**, and **csrSortedColIndA**); **f** and **x** are the right-hand-side and the solution vectors; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation $\mathrm{op}(A)$
m	number of rows and columns of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix types are **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrSortedValA	<type> array of **nnz** (= **csrSortedRowPtrA(m)** – **csrSortedRowPtrA(0)**) nonzero elements of matrix **A**.
csrSortedRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrSortedColIndA	integer array of **nnz** (= **csrSortedRowPtrA(m)** – **csrSortedRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
f	<type> right-hand-side vector of size **m**.

Output

x	<type> solution vector of size **m**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.16. cusparseCsrsv_solveEx()

```
cusparseStatus_t cusparseCsrsv_solveEx(cusparseHandle_t handle,
                                       cusparseOperation_t transA,
                                       int m,
                                       const void *alpha, cudaDataType
 alphatype,
                                       const cusparseMatDescr_t descrA,
                                       const void *csrSortedValA,
 cudaDataType csrSortedValAtype,
                                       const int *csrSortedRowPtrA,
                                       const int *csrSortedColIndA,
                                       cusparseSolveAnalysisInfo_t
 info,
                                       const void *f, cudaDataType
 ftype,
                                       void *x, cudaDataType xtype,
                                       cudaDataType executiontype);
```

This function is an extended version of **cusparse<t>csrsv_solve()**. For detailed description of the functionality, see **cusparse<t>csrsv_solve()**.

This function does not support half-precision execution type, but it supports half-precision IO with single precision execution.

Input specifically required by cusparseCsrsv_solveEx

alphatype	Data type of **alpha**.
csrSortedValAtype	Data type of **csrSortedValA**.
ftype	Data type of **f**.
xtype	Data type of **x**.
executiontype	Data type used for computation.

7.17. cusparse<t>csrsv2_bufferSize()

```
cusparseStatus_t
cusparseScsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseOperation_t transA,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           float *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDcsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseOperation_t transA,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           double *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCcsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseOperation_t transA,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           cuComplex *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrsv2Info_t info,
                           int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZcsrsv2_bufferSize(cusparseHandle_t handle,
                           cusparseOperation_t transA,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrsv2Info_t info,
                           int *pBufferSizeInBytes);
```

This function returns the size of the buffer used in **csrsv2**, a new sparse triangular linear system **op(A)*y** $= \alpha$ **x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of the parameter **trans** and the upper (lower) triangular part of **A**, **csrsv2_bufferSize()** returns the maximum size buffer of these combinations. The buffer size depends on the dimension and the number of nonzero elements of the matrix. If the user changes the matrix, it is necessary to call **csrsv2_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record of internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in the **csrsv2_analysis** and **csrsv2_solve**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.18. cusparse<t>csrsv2_analysis()

```
cusparseStatus_t
cusparseScsrsv2_analysis(cusparseHandle_t handle,
                         cusparseOperation_t transA,
                         int m,
                         int nnz,
                         const cusparseMatDescr_t descrA,
                         const float *csrValA,
                         const int *csrRowPtrA,
                         const int *csrColIndA,
                         csrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseDcsrsv2_analysis(cusparseHandle_t handle,
                         cusparseOperation_t transA,
                         int m,
                         int nnz,
                         const cusparseMatDescr_t descrA,
                         const double *csrValA,
                         const int *csrRowPtrA,
                         const int *csrColIndA,
                         csrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseCcsrsv2_analysis(cusparseHandle_t handle,
                         cusparseOperation_t transA,
                         int m,
                         int nnz,
                         const cusparseMatDescr_t descrA,
                         const cuComplex *csrValA,
                         const int *csrRowPtrA,
                         const int *csrColIndA,
                         csrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);

cusparseStatus_t
cusparseZcsrsv2_analysis(cusparseHandle_t handle,
                         cusparseOperation_t transA,
                         int m,
                         int nnz,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex *csrValA,
                         const int *csrRowPtrA,
                         const int *csrColIndA,
                         csrsv2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer);
```

This function performs the analysis phase of **csrsv2**, a new sparse triangular linear system **op(A)*y** = α **x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **csrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrsv2_analysis()** reports a structural zero and computes level information that is stored in opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **csrsv2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **csrsv2_analysis()** always reports the first structural zero, even if the policy is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is missing for some **j**. The user needs to call **cusparseXcsrsv2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **csrsv2_solve()** if **csrsv2_analysis()** reports a structural zero. In this case, the user can still call **csrsv2_solve()** which will return a numerical zero in the same position as the structural zero. However the result **x** is meaningless.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\text{op}(A)$.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** – **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** – **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

info	structure initialized using **cusparseCreateCsrsv2Info()**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrsv2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.19. cusparse<t>csrsv2_solve()

```
cusparseStatus_t
cusparseScsrsv2_solve(cusparseHandle_t handle,
                      cusparseOperation_t transA,
                      int m,
                      int nnz,
                      const float *alpha,
                      const cusparseMatDescr_t descra,
                      const float *csrValA,
                      const int *csrRowPtrA,
                      const int *csrColIndA,
                      csrsv2Info_t info,
                      const float *x,
                      float *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseDcsrsv2_solve(cusparseHandle_t handle,
                      cusparseOperation_t transA,
                      int m,
                      int nnz,
                      const double *alpha,
                      const cusparseMatDescr_t descra,
                      const double *csrValA,
                      const int *csrRowPtrA,
                      const int *csrColIndA,
                      csrsv2Info_t info,
                      const double *x,
                      double *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseCcsrsv2_solve(cusparseHandle_t handle,
                      cusparseOperation_t transA,
                      int m,
                      int nnz,
                      const cuComplex *alpha,
                      const cusparseMatDescr_t descra,
                      const cuComplex *csrValA,
                      const int *csrRowPtrA,
                      const int *csrColIndA,
                      csrsv2Info_t info,
                      const cuComplex *x,
                      cuComplex *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);

cusparseStatus_t
cusparseZcsrsv2_solve(cusparseHandle_t handle,
                      cusparseOperation_t transA,
                      int m,
                      int nnz,
                      const cuDoubleComplex *alpha,
                      const cusparseMatDescr_t descra,
                      const cuDoubleComplex *csrValA,
                      const int *csrRowPtrA,
                      const int *csrColIndA,
                      csrsv2Info_t info,
                      const cuDoubleComplex *x,
                      cuDoubleComplex *y,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer);
```

This function performs the solve phase of **csrsv2**, a new sparse triangular linear system **op(A)*y = α x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires the buffer size returned by **csrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **csrsv2_solve()** can be done without level information, the user still needs to be aware of consistency. If **csrsv2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrsv2_solve()** can be run with or without levels. On the contrary, if **csrsv2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrsv2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance but spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **csrsv2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **csrsv2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, picking faster one to perform the remaining iterations.

Function **csrsv2_solve()** reports the first numerical zero, including a structural zero. If **status** is 0, no numerical zero was found. Furthermore, no numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is zero for some **j**. The user needs to call **cusparseXcsrsv2_zeroPivot()** to know where the numerical zero is.

For example, suppose L is a lower triangular matrix with unit diagonal, the following code solves **L*y=x** by level information.

```
// Suppose that L is m x m sparse matrix represented by CSR format,
// L is lower triangular with unit diagonal.
// Assumption:
// - dimension of matrix L is m,
// - matrix L has nnz number zero elements,
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of L on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.

cusparseMatDescr_t descr = 0;
csrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal, specified by parameter CUSPARSE_DIAG_TYPE_UNIT
//   (L may not have all diagonal elements.)
cusparseCreateMatDescr(&descr);
cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatFillMode(descr, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr, CUSPARSE_DIAG_TYPE_UNIT);

// step 2: create a empty info structure
cusparseCreateCsrsv2Info(&info);

// step 3: query how much memory used in csrsv2, and allocate the buffer
cusparseDcsrsv2_bufferSize(handle, trans, m, nnz, descr,
    d_csrVal, d_csrRowPtr, d_csrColInd, &pBufferSize);
// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis
cusparseDcsrsv2_analysis(handle, trans, m, nnz, descr,
    d_csrVal, d_csrRowPtr, d_csrColInd,
    info, policy, pBuffer);
// L has unit diagonal, so no structural zero is reported.
status = cusparseXcsrsv2_zeroPivot(handle, info, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is missing\n", structural_zero, structural_zero);
}

// step 5: solve L*y = x
cusparseDcsrsv2_solve(handle, trans, m, nnz, &alpha, descr,
   d_csrVal, d_csrRowPtr, d_csrColInd, info,
   d_x, d_y, policy, pBuffer);
// L has unit diagonal, so no numerical zero is reported.
status = cusparseXcsrsv2_zeroPivot(handle, info, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is zero\n", numerical_zero, numerical_zero);
}

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyCsrsv2Info(info);
cusparseDestroyMatDescr(descr);
cusparseDestroy(handle);
```

Remark: **csrsv2_solve()** needs more nonzeros per row to achieve good performance. It would perform better if more than 16 nonzeros per row in average.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$.
m	number of rows and columns of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** – **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** – **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
x	<type> right-hand-side vector of size **m**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is return by **csrsv2_bufferSize**.

Output

y	<type> solution vector of size **m**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.20. cusparseXcsrsv2_zeroPivot()

```
cusparseStatus_t
cusparseXcsrsv2_zeroPivot(cusparseHandle_t handle,
                          csrsv2Info_t info,
                          int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero. Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrsv2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrsv2_analysis()** or **csrsv2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

7.21. cusparse<t>hybmv()

```
cusparseStatus_t
cusparseShybmv(cusparseHandle_t handle, cusparseOperation_t transA,
               const float           *alpha,
               const cusparseMatDescr_t descrA,
               const cusparseHybMat_t hybA, const float           *x,
               const float           *beta, float           *y)
cusparseStatus_t
cusparseDhybmv(cusparseHandle_t handle, cusparseOperation_t transA,
               const double          *alpha,
               const cusparseMatDescr_t descrA,
               const cusparseHybMat_t hybA, const double          *x,
               const double          *beta, double          *y)
cusparseStatus_t
cusparseChybmv(cusparseHandle_t handle, cusparseOperation_t transA,
               const cuComplex       *alpha,
               const cusparseMatDescr_t descrA,
               const cusparseHybMat_t hybA, const cuComplex       *x,
               const cuComplex       *beta, cuComplex       *y)
cusparseStatus_t
cusparseZhybmv(cusparseHandle_t handle, cusparseOperation_t transA,
               const cuDoubleComplex *alpha,
               const cusparseMatDescr_t descrA,
               const cusparseHybMat_t hybA, const cuDoubleComplex *x,
               const cuDoubleComplex *beta, cuDoubleComplex *y)
```

This function performs the matrix-vector operation

$$y = \alpha * op(A) * x + \beta * y$$

A is an **m×n** sparse matrix that is defined in the HYB storage format by an opaque data structure **hybA**, **x** and **y** are vectors, α and β are scalars, and

$$op(A) = \{A \quad \text{if transA} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE}$$

Notice that currently only $op(A) = A$ is supported.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $op(A)$ (currently only $op(A) = A$ is supported).
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.
hybA	the matrix **A** in HYB storage format.
x	<type> vector of **n** elements.

beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of **m** elements.

Output

y	<type> updated vector.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	the internally stored HYB format parameters are invalid.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.22. cusparse<t>hybsv_analysis()

```
cusparseStatus_t
cusparseShybsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        const cusparseMatDescr_t descrA,
                        cusparseHybMat_t hybA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseDhybsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        const cusparseMatDescr_t descrA,
                        cusparseHybMat_t hybA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseChybsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        const cusparseMatDescr_t descrA,
                        cusparseHybMat_t hybA,
                        cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseZhybsv_analysis(cusparseHandle_t handle,
                        cusparseOperation_t transA,
                        const cusparseMatDescr_t descrA,
                        cusparseHybMat_t hybA,
                        cusparseSolveAnalysisInfo_t info)
```

This function performs the analysis phase of the solution of a sparse triangular linear system

$$\mathrm{op}(A) * \mathrm{y} = \alpha * \mathrm{x}$$

A is an **m×m** sparse matrix that is defined in HYB storage format by an opaque data structure **hybA**, **x** and **y** are the right-hand-side and the solution vectors, α is a scalar, and

$$\mathrm{op}(A) = \{ A \quad \text{if transA} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE}$$

Notice that currently only $\mathrm{op}(A) = A$ is supported.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a significant amount of extra storage that is proportional to the matrix size. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$ (currently only $\mathrm{op}(A) = A$ is supported).
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and diagonal type **USPARSE_DIAG_TYPE_NON_UNIT**.
hybA	the matrix **A** in HYB storage format.
info	structure initialized using **cusparseCreateSolveAnalysisInfo()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	the internally stored HYB format parameters are invalid.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.23. cusparse<t>hybsv_solve()

```
cusparseStatus_t
cusparseShybsv_solve(cusparseHandle_t handle,
                     cusparseOperation_t transA,
                     const float           *alpha,
                     const cusparseMatDescr_t descrA,
                     cusparseHybMat_t hybA,
                     cusparseSolveAnalysisInfo_t info,
                     const float           *x, float           *y)
cusparseStatus_t
cusparseDhybsv_solve(cusparseHandle_t handle,
                     cusparseOperation_t transA,
                     const double          *alpha,
                     const cusparseMatDescr_t descrA,
                     cusparseHybMat_t hybA,
                     cusparseSolveAnalysisInfo_t info,
                     const double          *x, double          *y)
cusparseStatus_t
cusparseChybsv_solve(cusparseHandle_t handle,
                     cusparseOperation_t transA,
                     const cuComplex       *alpha,
                     const cusparseMatDescr_t descrA,
                     cusparseHybMat_t hybA,
                     cusparseSolveAnalysisInfo_t info,
                     const cuComplex       *x, cuComplex       *y)
cusparseStatus_t
cusparseZhybsv_solve(cusparseHandle_t handle,
                     cusparseOperation_t transA,
                     const cuDoubleComplex *alpha,
                     const cusparseMatDescr_t descrA,
                     cusparseHybMat_t hybA,
                     cusparseSolveAnalysisInfo_t info,
                     const cuDoubleComplex *x, cuDoubleComplex *y)
```

This function performs the solve phase of the solution of a sparse triangular linear system:

$$\mathrm{op}(A) * \mathrm{y} = \alpha * \mathrm{x}$$

A is an **m×m** sparse matrix that is defined in HYB storage format by an opaque data structure **hybA**, **x** and **y** are the right-hand-side and the solution vectors, α is a scalar, and

$$\mathrm{op}(A) = \{A \quad \text{if transA} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE}$$

Notice that currently only $\mathrm{op}(A) = A$ is supported.

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.

transA	the operation $\mathrm{op}(A)$ (currently only $\mathrm{op}(A) = A$ is supported).
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and the diagonal type is **CUSPARSE_DIAG_TYPE_NON_UNIT**.
hybA	the matrix **A** in HYB storage format.
info	structure with information collected during the analysis phase (that should be passed to the solve phase unchanged).
x	<type> right-hand-side vector of size **m**.

Output

y	<type> solution vector of size **m**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the internally stored hyb format parameters are invalid.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

7.24. cusparseCsr2cscEx2()

```
cusparseStatus_t
cusparseCsr2cscEx2(cusparseHandle_t     handle,
                   int                  m,
                   int                  n,
                   int                  nnz,
                   const void*          csrVal,
                   const int*           csrRowPtr,
                   const int*           csrColInd,
                   void*                cscVal,
                   int*                 cscColPtr,
                   int*                 cscRowInd,
                   cudaDataType         valType,
                   cusparseAction_t     copyValues,
                   cusparseIndexBase_t  idxBase,
                   cusparseCsr2CscAlg_t alg,
                   void*                buffer)
```

This function converts a sparse matrix in CSR format (that is defined by the three arrays **csrVal**, **csrRowPtr**, and **csrColInd**) into a sparse matrix in CSC format (that is defined by arrays **cscVal**, **cscRowInd**, and **cscColPtr**). The resulting matrix can also be seen as the transpose of the original sparse matrix. Notice that this routine can also be used to convert a matrix in CSC format into a matrix in CSR format.

For alg **CUSPARSE_CSR2CSC_ALG1**: it requires extra storage proportional to the number of nonzero values **nnz**. It provides in output always the same matrix.

For alg **CUSPARSE_CSR2CSC_ALG2**: it requires extra storage proportional to the number of rows **m**. It does not ensure always the same ordering of CSC column indices and values. Also, it provides better performance then **CUSPARSE_CSR2CSC_ALG1** for regular matrices.

It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context
m	number of rows of the CSR input matrix; number of columns of the CSC ouput matrix
n	number of columns of the CSR input matrix; number of rows of the CSC ouput matrix
nnz	number of nonzero elements of the CSR and CSC matrices
csrVal	value array of size **nnz** of the CSR matrix; of same type as **valType**
csrRowPtr	integer array of size **m + 1** that containes the CSR row offsets
csrColInd	integer array of size **nnz** that containes the CSR column indices
valType	value type for both CSR and CSC matrices
copyValues	**CUSPARSE_ACTION_SYMBOLIC** or **CUSPARSE_ACTION_NUMERIC**
idxBase	Index base **CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.
alg	algorithm implementation. see **cusparseCsr2CscAlg_t** for possible values.
buffer	pointer to workspace buffer

Output

cscVal	value array of size **nnz** of the CSC matrix; of same type as **valType**
cscColPtr	integer array of size **n + 1** that containes the CSC column offsets
cscRowInd	integer array of size **nnz** that containes the CSC row indices

7.25. cusparseCsr2cscEx2_bufferSize()

```
cusparseStatus_t
cusparseCsr2cscEx2_bufferSize(cusparseHandle_t     handle,
                              int                  m,
                              int                  n,
                              int                  nnz,
                              const void*          csrVal,
                              const int*           csrRowPtr,
                              const int*           csrColInd,
                              void*                cscVal,
                              int*                 cscColPtr,
                              int*                 cscRowInd,
                              cudaDataType         valType,
                              cusparseAction_t     copyValues,
                              cusparseIndexBase_t  idxBase,
                              cusparseCsr2CscAlg_t alg,
                              size_t*              bufferSize)
```

This function returns the size of the workspace needed by **cusparseCsr2cscEx2()**. User needs to allocate a buffer of this size and give that buffer to **cusparseCsr2cscEx2()** as an argument. All the arguments are similar to **cusparseCsr2cscEx2()** except the following output argument.

Output

bufferSize	number of bytes of workspace needed by **cusparseCsr2cscEx2().**

Chapter 8. CUSPARSE LEVEL 3 FUNCTION REFERENCE

This chapter describes sparse linear algebra functions that perform operations between sparse and (usually tall) dense matrices.

In particular, the solution of sparse triangular linear systems with multiple right-hand sides is implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate **`csrsm_analysis()`** function. The analysis is specific to the sparsity pattern of the given matrix and to the selected **`cusparseOperation_t`** type. The information from the analysis phase is stored in the parameter of type **`cusparseSolveAnalysisInfo_t`** that has been initialized previously with a call to **`cusparseCreateSolveAnalysisInfo()`**.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the **`cusparseSolveAnalysisInfo_t`** parameter by calling the appropriate **`csrsm_solve()`** function. The solve phase may be performed multiple times with different multiple right-hand sides, while the analysis phase needs to be performed only once. This is especially useful when a sparse triangular linear system must be solved for different sets of multiple right-hand sides one at a time, while its coefficient matrix remains the same.

Finally, once all the solves have completed, the opaque data structure pointed to by the **`cusparseSolveAnalysisInfo_t`** parameter can be released by calling **`cusparseDestroySolveAnalysisInfo()`**. For more information please refer to [3].

8.1. cusparse<t>csrmm()

```
cusparseStatus_t
cusparseScsrmm(cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    int k,
    int nnz,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const float *B,
    int ldb,
    const float *beta,
    float *C,
    int ldc)
cusparseStatus_t
cusparseDcsrmm(cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    int k,
    int nnz,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const double *B,
    int ldb,
    const double *beta,
    double *C,
    int ldc)
cusparseStatus_t
cusparseCcsrmm(cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    int k,
    int nnz,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cuComplex *B,
    int ldb,
    const cuComplex *beta,
    cuComplex *C,
    int ldc)
cusparseStatus_t
cusparseZcsrmm(cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    int k,
    int nnz,
    const cuDoubleComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cuDoubleComplex *B,
```

This function performs one of the following matrix-matrix operations:

$$C = \alpha * \mathrm{op}(A) * B + \beta * C$$

A is an **m×k** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **C** are dense matrices; α and β are scalars; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

When using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this routine may produce slightly different results with the same input parameters during different runs of this function. For these matrix types it uses atomic operations to compute the final result; consequently, many threads may be adding floating point numbers to the same memory location without any specific ordering, which may produce slightly different results for each run.

If exactly the same output is required for any input when multiplying by the transpose of a general matrix, the following procedure can be used:

1. Convert the matrix from CSR to CSC format using one of the **csr2csc()** functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.

2. Call the **csrmm()** function with the **cusparseOperation_t** parameter set to **CUSPARSE_OPERATION_NON_TRANSPOSE** and with the interchanged rows and columns of the matrix stored in CSC format. This (implicitly) multiplies the vector by the transpose of the matrix in the original CSR format.

This function requires no extra storage for the general matrices when operation **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It requires some extra storage for Hermitian/symmetric matrices and for the general matrices when an operation different from **CUSPARSE_OPERATION_NON_TRANSPOSE** is selected. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$
m	number of rows of sparse matrix **A**.
n	number of columns of dense matrices **B** and **C**.
k	number of columns of sparse matrix **A**.
nnz	number of nonzero elements of sparse matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix types are **CUSPARSE_MATRIX_TYPE_GENERAL**, **CUSPARSE_MATRIX_TYPE_SYMMETRIC**, and **CUSPARSE_MATRIX_TYPE_HERMITIAN**. Also, the supported index bases are

	CUSPARSE_INDEX_BASE_ZERO and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
B	array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. It must be at least $\max(1, k)$ if $\text{op}(A) = A$ and at least $\max(1, m)$ otherwise.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least $\max(1, m)$ if $\text{op}(A) = A$ and at least $\max(1, k)$ otherwise.

Output

C	<type> updated array of dimensions **(ldc, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n, k, nnz<0** or **ldb** and **ldc** are incorrect).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.2. cusparse<t>csrmm2()

```
cusparseStatus_t
cusparseScsrmm2(cusparseHandle_t         handle,
                cusparseOperation_t      transA,
                cusparseOperation_t      transB,
                int                      m,
                int                      n,
                int                      k,
                int                      nnz,
                const float              *alpha,
                const cusparseMatDescr_t descrA,
                const float              *csrValA,
                const int                *csrRowPtrA,
                const int                *csrColIndA,
                const float              *B,
                int                      ldb,
                const float              *beta,
                float                    *C,
                int                      ldc)
cusparseStatus_t
cusparseDcsrmm2(cusparseHandle_t         handle,
                cusparseOperation_t      transA,
                cusparseOperation_t      transB,
                int                      m,
                int                      n,
                int                      k,
                int                      nnz,
                const double             *alpha,
                const cusparseMatDescr_t descrA,
                const double             *csrValA,
                const int                *csrRowPtrA,
                const int                *csrColIndA,
                const double             *B,
                int                      ldb,
                const double             *beta,
                double                   *C,
                int                      ldc)
cusparseStatus_t
cusparseCcsrmm2(cusparseHandle_t         handle,
                cusparseOperation_t      transA,
                cusparseOperation_t      transB,
                int                      m,
                int                      n,
                int                      k,
                int                      nnz,
                const cuComplex          *alpha,
                const cusparseMatDescr_t descrA,
                const cuComplex          *csrValA,
                const int                *csrRowPtrA,
                const int                *csrColIndA,
                const cuComplex          *B,
                int                      ldb,
                const cuComplex          *beta,
                cuComplex                *C,
                int                      ldc)
cusparseStatus_t
cusparseZcsrmm2(cusparseHandle_t         handle,
                cusparseOperation_t      transA,
                cusparseOperation_t      transB,
                int                      m,
                int                      n,
                int                      k,
                int                      nnz,
                const cuDoubleComplex    *alpha,
                const cusparseMatDescr_t descrA,
```

This function performs one of the following matrix-matrix operations:

$$C = \alpha * \mathrm{op}(A) * \mathrm{op}(B) + \beta * C$$

A is an **m×k** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **C** are dense matrices; α and β are scalars; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$\mathrm{op}(B) = \begin{cases} B & \text{if transB == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T & \text{if transB == CUSPARSE\_OPERATION\_TRANSPOSE} \\ B^H & \text{not supported} \end{cases}$$

If **op(B)=B**, **cusparse<t>csrmm2()** is the same as **cusparse<t>csrmm()**; otherwise, only **op(A)=A** is supported and the matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**.

The motivation of **transpose(B)** is to improve the memory access of matrix **B**. The computational pattern of **A*transpose(B)** with matrix **B** in column-major order is equivalent to **A*B** with matrix **B** in row-major order.

In practice, no operation in iterative solver or eigenvalue solver uses **A*transpose(B)**. However we can perform **A*transpose(transpose(B))** which is the same as **A*B**. For example, suppose **A** is **m*k**, **B** is **k*n** and **C** is **m*n**, the following code shows usage of **cusparseDcsrmm()**.

```
// A is m*k, B is k*n and C is m*n
    const int ldb_B = k; // leading dimension of B
    const int ldc   = m; // leading dimension of C
// perform C:=alpha*A*B + beta*C
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );
    cusparseDcsrmm(cusparse_handle,
                CUSPARSE_OPERATION_NON_TRANSPOSE,
                m, n, k, nnz, alpha,
                descrA, csrValA, csrRowPtrA, csrColIndA,
                B, ldb_B,
                beta, C, ldc);
```

Instead of using **A*B**, our proposal is to transpose **B** to **Bt** first by calling **cublas<t>geam()**, then to perform **A*transpose(Bt)**.

```
// step 1: Bt := transpose(B)
    double *Bt;
    const int ldb_Bt = n; // leading dimension of Bt
    cudaMalloc((void**)&Bt, sizeof(double)*ldb_Bt*k);
    double one  = 1.0;
    double zero = 0.0;
    cublasSetPointerMode(cublas_handle, CUBLAS_POINTER_MODE_HOST);
    cublasDgeam(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_T,
        n, k, &one, B, int ldb_B, &zero, B, int ldb_B, Bt, ldb_Bt);

// step 2: perform C:=alpha*A*transpose(Bt) + beta*C
    cusparseDcsrmm2(cusparse_handle,
                CUSPARSE_OPERATION_NON_TRANSPOSE,
                CUSPARSE_OPERATION_TRANSPOSE
                m, n, k, nnz, alpha,
                descrA, csrValA, csrRowPtrA, csrColIndA,
                Bt, ldb_Bt,
                beta, C, ldc);
```

Remark 1: **cublas<t>geam()** and **cusparse<t>csrmm2()** are memory bound. The complexity of **cublas<t>geam()** is **2*n*k**, and the minimum complexity of **cusparse<t>csrmm2()** is about **(nnz + nnz*n + 2*m*n)**. If **nnz** per column **(=nnz/k)** is large, it is worth paying the extra cost on transposition because **A*transpose(B)** may be 2× faster than **A*B** if the sparsity pattern of **A** is not good.

Remark 2: **A*transpose(B)** is only supported on compute capability 2.0 and above.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$
transB	the operation $\mathrm{op}(B)$
m	number of rows of sparse matrix **A**.
n	number of columns of dense matrix $\mathrm{op}(B)$ and **C**.
k	number of columns of sparse matrix **A**.
nnz	number of nonzero elements of sparse matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix types is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

B	array of dimensions **(ldb, n)** if **op(B)=B** and **(ldb, k)** otherwise.
ldb	leading dimension of **B**. If **op(B)=B**, it must be at least $\max(1, k)$ if $\text{op}(A) = A$ and at least $\max(1, m)$ otherwise. If **op(B)!=B**, it must be at least **max(1, n)**.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least $\max(1, m)$ if $\text{op}(A) = A$ and at least $\max(1, k)$ otherwise.

Output

C	<type> updated array of dimensions **(ldc, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n, k, nnz<0** or **ldb** and **ldc** are incorrect).
CUSPARSE_STATUS_ARCH_MISMATCH	if **op(B)=B**, the device does not support double precision or if **op(B)=transpose(B)** the device is below compute capability 2.0.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported otherwise.

8.3. cusparse<t>csrsm_analysis()

```
cusparseStatus_t
cusparseScsrsm_analysis(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info)


cusparseStatus_t
cusparseDcsrsm_analysis(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info)


cusparseStatus_t
cusparseCcsrsm_analysis(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info)


cusparseStatus_t
cusparseZcsrsm_analysis(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int nnz,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info)
```

This function performs the analysis phase of the solution of a sparse triangular linear system

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

with multiple right-hand sides, where **A** is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**; **B** and **X** are the right-hand-side and the solution dense matrices; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a significant amount of extra storage that is proportional to the matrix size. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix types are **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) – csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) – csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure initialized using **cusparseCreateSolveAnalysisInfo()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.

CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.4. cusparse<t>csrsm_solve()

```
cusparseStatus_t
cusparseScsrsm_solve(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info,
    const float *B,
    int ldb,
    float *X,
    int ldx)

cusparseStatus_t
cusparseDcsrsm_solve(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info,
    const double *B,
    int ldb,
    double *X,
    int ldx)

cusparseStatus_t
cusparseCcsrsm_solve(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cusparseSolveAnalysisInfo_t info,
    const cuComplex *B,
    int ldb,
    cuComplex *X,
    int ldx)

cusparseStatus_t
cusparseZcsrsm_solve(
    cusparseHandle_t handle,
    cusparseOperation_t transA,
    int m,
    int n,
    const cuDoubleComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
```

This function performs the solve phase of the solution of a sparse triangular linear system

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

with multiple right-hand sides, where **A** is an **m×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side and the solution dense matrices; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
transA	the operation **op(A)**.
m	number of rows and columns of matrix **A**.
n	number of columns of matrix **X** and **Y**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix types are **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should be passed to the solve phase unchanged).
B	<type> right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B** (that is ≥ $\max(1, \mathrm{m})$).

Output

X	<type> solution array of dimensions **(ldx, n)**.
ldx	leading dimension of **X** (that is ≥ $\max(1, \mathrm{m})$).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (`m<0`).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.5. cusparse<t>csrsm2_bufferSizeExt()

```
cusparseStatus_t
cusparseScsrsm2_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const float *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    size_t *pBufferSize)

cusparseStatus_t
cusparseDcsrsm2_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const double *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    size_t *pBufferSize)

cusparseStatus_t
cusparseCcsrsm2_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const cuComplex *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    size_t *pBufferSize)

cusparseStatus_t
cusparseZcsrsm2_bufferSizeExt(
```

This function returns the size of the buffer used in **csrsm2**, a sparse triangular linear system **op(A) * op(X) =** α **op(B)**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Input

handle	handle to the cuSPARSE library context.
algo	algo = 0 is non-block version; algo = 1 is block version.
transA	the operation $\text{op}(A)$.
transB	the operation $\text{op}(B)$.
m	number of rows of matrix **A**.
nrhs	number of columns of right hand side matrix **op(B)**.
nnz	number of nonzero elements of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
B	<type> right-hand-side matrix. **op(B)** is of size **m-by-nrhs**.
ldb	leading dimension of **B** and **X**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.

Output

info	record of internal states based on different algorithms.

pBufferSize	number of bytes of the buffer used in the **csrsm2_analysis** and **csrsm2_solve**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

8.6. cusparse<t>csrsm2_analysis()

```
cusparseStatus_t
cusparseScsrsm2_analysis(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const float *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseDcsrsm2_analysis(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const double *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseCcsrsm2_analysis(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const cuComplex *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseZcsrsm2_analysis(
```

This function performs the analysis phase of **csrsm2**, a sparse triangular linear system **op(A) * op(X) =** α **op(B)**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **csrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrsm2_analysis()** reports a structural zero and computes level information that is stored in opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **csrsm2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **csrsm2_analysis()** always reports the first structural zero, even if the policy is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is missing for some **j**. The user needs to call **cusparseXcsrsm2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **csrsm2_solve()** if **csrsm2_analysis()** reports a structural zero. In this case, the user can still call **csrsm2_solve()** which will return a numerical zero in the same position as the structural zero. However the result **X** is meaningless.

Input

handle	handle to the cuSPARSE library context.
algo	algo = 0 is non-block version; algo = 1 is block version.
transA	the operation $\text{op}(A)$.
transB	the operation $\text{op}(B)$.
m	number of rows of matrix **A**.
nrhs	number of columns of matrix **op(B)**.
nnz	number of nonzero elements of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.

csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
B	<type> right-hand-side matrix. **op(B)** is of size **m-by-nrhs**.
ldb	leading dimension of **B** and **X**.
info	structure initialized using **cusparseCreateCsrsv2Info()**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrsm2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**).
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

8.7. cusparse<t>csrsm2_solve()

```
cusparseStatus_t
cusparseScsrsm2_solve(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    float *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseDcsrsm2_solve(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    double *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseCcsrsm2_solve(
    cusparseHandle_t handle,
    int algo,
    cusparseOperation_t transA,
    cusparseOperation_t transB,
    int m,
    int nrhs,
    int nnz,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    cuComplex *B,
    int ldb,
    csrsm2Info_t info,
    cusparseSolvePolicy_t policy,
    void *pBuffer)

cusparseStatus_t
cusparseZcsrsm2_solve(
```

This function performs the solve phase of **csrsm2**, a sparse triangular linear system **op(A) * op(X) =** α **op(B)**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

transB acts on both matrix **B** and matrix **X**, only **CUSPARSE_OPERATION_NON_TRANSPOSE** and **CUSPARSE_OPERATION_TRANSPOSE**. The operation is in-place, matrix **B** is overwritten by matrix **X**.

ldb must be not less than **m** if **transB** = **CUSPARSE_OPERATION_NON_TRANSPOSE**. Otherwise, **ldb** must be not less than **nrhs**.

This function requires the buffer size returned by **csrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **csrsm2_solve()** can be done without level information, the user still needs to be aware of consistency. If **csrsm2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrsm2_solve()** can be run with or without levels. On the contrary, if **csrsm2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrsm2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance but spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **csrsm2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **csrsm2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, picking faster one to perform the remaining iterations.

Function **csrsm2_solve()** reports the first numerical zero, including a structural zero. If **status** is 0, no numerical zero was found. Furthermore, no numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is zero for some **j**. The user needs to call **cusparseXcsrsm2_zeroPivot()** to know where the numerical zero is.

csrsm2 provides two algorithms specified by the parameter **algo**. **algo=0** is non-block version and **algo=1** is block version. non-block version is memory-bound, limited by bandwidth. block version partitions the matrix into small tiles and applies desne operations. Although it has more flops than non-block version, it may be faster if non-block version already reaches maximum bandwidth..

Appendix H shows an example of csrsm2.

Input

handle	handle to the cuSPARSE library context.

algo	algo = 0 is non-block version; algo = 1 is block version.
transA	the operation $\mathrm{op}(A)$.
transB	the operation $\mathrm{op}(B)$.
m	number of rows and columns of matrix **A**.
nrhs	number of columns of matrix **op(B)**.
nnz	number of nonzeros of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
B	<type> right-hand-side matrix. **op(B)** is of size **m-by-nrhs**.
ldb	leading dimension of **B** and **X**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrsm2_bufferSize**.

Output

X	<type> solution matrix, **op(X)** is of size **m-by-nrhs**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**).
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

8.8. cusparseXcsrsm2_zeroPivot()

```
cusparseStatus_t
cusparseXcsrsm2_zeroPivot(cusparseHandle_t handle,
                          csrsm2Info_t info,
                          int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero. Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrsm2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrsm2_analysis()** or **csrsm2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

8.9. cusparse<t>bsrmm()

```
cusparseStatus_t
cusparseSbsrmm(cusparseHandle_t handle,
               cusparseDirection_t dirA,
               cusparseOperation_t transA,
               cusparseOperation_t transB,
               int mb,
               int n,
               int kb,
               int nnzb,
               const float *alpha,
               const cusparseMatDescr_t descrA,
               const float *bsrValA,
               const int *bsrRowPtrA,
               const int *bsrColIndA,
               const int  blockDim,
               const float *B,
               const int ldb,
               const float *beta,
               float *C,
               int ldc)

cusparseStatus_t
cusparseDbsrmm(cusparseHandle_t handle,
               cusparseDirection_t dirA,
               cusparseOperation_t transA,
               cusparseOperation_t transB,
               int mb,
               int n,
               int kb,
               int nnzb,
               const double *alpha,
               const cusparseMatDescr_t descrA,
               const double *bsrValA,
               const int *bsrRowPtrA,
               const int *bsrColIndA,
               const int  blockDim,
               const double *B,
               const int ldb,
               const double *beta,
               double *C,
               int ldc)

cusparseStatus_t
cusparseCbsrmm(cusparseHandle_t handle,
               cusparseDirection_t dirA,
               cusparseOperation_t transA,
               cusparseOperation_t transB,
               int mb,
               int n,
               int kb,
               int nnzb,
               const cuComplex *alpha,
               const cusparseMatDescr_t descrA,
               const cuComplex *bsrValA,
               const int *bsrRowPtrA,
               const int *bsrColIndA,
               const int  blockDim,
               const cuComplex *B,
               const int ldb,
               const cuComplex *beta,
               cuComplex *C,
               int ldc)
```

This function performs one of the following matrix-matrix operations:

$$C = \alpha * \mathrm{op}(A) * \mathrm{op}(B) + \beta * C$$

A is an **mb×kb** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**; **B** and **C** are dense matrices; α and β are scalars; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE (not supported)} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

and

$$\mathrm{op}(B) = \begin{cases} B & \text{if transB == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T & \text{if transB == CUSPARSE\_OPERATION\_TRANSPOSE} \\ B^H & \text{if transB == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**.

The motivation of **transpose(B)** is to improve memory access of matrix **B**. The computational pattern of **A*transpose(B)** with matrix **B** in column-major order is equivalent to **A*B** with matrix **B** in row-major order.

In practice, no operation in an iterative solver or eigenvalue solver uses **A*transpose(B)**. However, we can perform **A*transpose(transpose(B))** which is the same as **A*B**. For example, suppose **A** is **mb*kb**, **B** is **k*n** and **C** is **m*n**, the following code shows usage of **cusparseDbsrmm()**.

```
// A is mb*kb, B is k*n and C is m*n
    const int m = mb*blockSize;
    const int k = kb*blockSize;
    const int ldb_B = k; // leading dimension of B
    const int ldc   = m; // leading dimension of C
// perform C:=alpha*A*B + beta*C
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );
    cusparseDbsrmm(cusparse_handle,
                CUSPARSE_DIRECTION_COLUMN,
                CUSPARSE_OPERATION_NON_TRANSPOSE,
                CUSPARSE_OPERATION_NON_TRANSPOSE,
                mb, n, kb, nnzb, alpha,
                descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockSize,
                B, ldb_B,
                beta, C, ldc);
```

Instead of using **A*B**, our proposal is to transpose **B** to **Bt** by first calling **cublas<t>geam()**, and then to perform **A*transpose(Bt)**.

```
// step 1: Bt := transpose(B)
    const int m = mb*blockSize;
    const int k = kb*blockSize;
    double *Bt;
    const int ldb_Bt = n; // leading dimension of Bt
    cudaMalloc((void**)&Bt, sizeof(double)*ldb_Bt*k);
    double one  = 1.0;
    double zero = 0.0;
    cublasSetPointerMode(cublas_handle, CUBLAS_POINTER_MODE_HOST);
    cublasDgeam(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_T,
        n, k, &one, B, int ldb_B, &zero, B, int ldb_B, Bt, ldb_Bt);

// step 2: perform C:=alpha*A*transpose(Bt) + beta*C
    cusparseDbsrmm(cusparse_handle,
                CUSPARSE_DIRECTION_COLUMN,
                CUSPARSE_OPERATION_NON_TRANSPOSE,
                CUSPARSE_OPERATION_TRANSPOSE,
                mb, n, kb, nnzb, alpha,
                descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockSize,
                Bt, ldb_Bt,
                beta, C, ldc);
```

Function **bsrmm()** is only supported on compute capability 2.0 and above.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transB	the operation **op(B)**.
mb	number of block rows of sparse matrix **A**.
n	number of columns of dense matrix **op(B)** and **A**.
kb	number of block columns of sparse matrix **A**.
nnzb	number of non-zero blocks of sparse matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.

blockDim	block dimension of sparse matrix **A**, larger than zero.
B	array of dimensions **(ldb, n)** if **op(B)=B** and **(ldb, k)** otherwise.
ldb	leading dimension of **B**. If **op(B)=B**, it must be at least $\max(1, k)$ If **op(B) != B**, it must be at least **max(1, n)**.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least $\max(1, m)$ if **op(A)=A** and at least $\max(1, k)$ otherwise.

Output

C	<type> updated array of dimensions **(ldc, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	Either invalid parameters were passed (**mb, n, kb, nnzb<0**; or **ldb** and **ldc** are incorrect). Either invalid or unsupported operations were passed (**op(A)** is different from CUSPARSE_OPERATION_NON_TRANSPOSE, or **op(B)** is different from CUSPARSE_OPERATION_NON_TRANSPOSE or CUSPARSE_OPERATION_TRANSPOSE).
CUSPARSE_STATUS_ARCH_MISMATCH	if device is below compute capability 2.0.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported otherwise.

8.10. cusparse<t>bsrsm2_bufferSize()

```
cusparseStatus_t
cusparseSbsrsm2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           cusparseOperation_t transX,
                           int mb,
                           int n,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           float *bsrSortedValA,
                           const int *bsrSortedRowPtrA,
                           const int *bsrSortedColIndA,
                           int blockDim,
                           bsrsm2Info_t info,
                           int *pBufferSizeInBytes)

cusparseStatus_t
cusparseDbsrsm2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           cusparseOperation_t transX,
                           int mb,
                           int n,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           double *bsrSortedValA,
                           const int *bsrSortedRowPtrA,
                           const int *bsrSortedColIndA,
                           int blockDim,
                           bsrsm2Info_t info,
                           int *pBufferSizeInBytes)

cusparseStatus_t
cusparseCbsrsm2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           cusparseOperation_t transX,
                           int mb,
                           int n,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex *bsrSortedValA,
                           const int *bsrSortedRowPtrA,
                           const int *bsrSortedColIndA,
                           int blockDim,
                           bsrsm2Info_t info,
                           int *pBufferSizeInBytes)

cusparseStatus_t
cusparseZbsrsm2_bufferSize(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           cusparseOperation_t transA,
                           cusparseOperation_t transX,
                           int mb,
                           int n,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex *bsrSortedValA,
                           const int *bsrSortedRowPtrA,
                           const int *bsrSortedColIndA,
                           int blockDim,
                           bsrsm2Info_t info,
                           int *pBufferSizeInBytes)
```

This function returns size of buffer used in **bsrsm2()**, a new sparse triangular linear system **op(A)*op(X)** = α **op(B)**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of parameter **trans** and the upper (and lower) triangular part of **A**, **bsrsm2_bufferSize()** returns the maximum size of the buffer among these combinations. The buffer size depends on dimension **mb,blockDim** and the number of nonzeros of the matrix, **nnzb**. If the user changes the matrix, it is necessary to call **bsrsm2_bufferSize()** again to get the correct buffer size, otherwise a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transX	the operation **op(X)**.
mb	number of block rows of matrix **A**.
n	number of columns of matrix **op(B)** and **op(X)**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**; larger than zero.

Output

info	record internal states based on different algorithms.

pBufferSizeInBytes	number of bytes of the buffer used in **bsrsm2_analysis()** and **bsrsm2_solve()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb, n, nnzb<=0**); base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.11. cusparse<t>bsrsm2_analysis()

```
cusparseStatus_t
cusparseSbsrsm2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         cusparseOperation_t transX,
                         int mb,
                         int n,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const float *bsrSortedVal,
                         const int *bsrSortedRowPtr,
                         const int *bsrSortedColInd,
                         int blockDim,
                         bsrsm2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer)

cusparseStatus_t
cusparseDbsrsm2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         cusparseOperation_t transX,
                         int mb,
                         int n,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const double *bsrSortedVal,
                         const int *bsrSortedRowPtr,
                         const int *bsrSortedColInd,
                         int blockDim,
                         bsrsm2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer)

cusparseStatus_t
cusparseCbsrsm2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         cusparseOperation_t transX,
                         int mb,
                         int n,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuComplex *bsrSortedVal,
                         const int *bsrSortedRowPtr,
                         const int *bsrSortedColInd,
                         int blockDim,
                         bsrsm2Info_t info,
                         cusparseSolvePolicy_t policy,
                         void *pBuffer)

cusparseStatus_t
cusparseZbsrsm2_analysis(cusparseHandle_t handle,
                         cusparseDirection_t dirA,
                         cusparseOperation_t transA,
                         cusparseOperation_t transX,
                         int mb,
                         int n,
                         int nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex *bsrSortedVal,
                         const int *bsrSortedRowPtr,
                         const int *bsrSortedColInd,
```

This function performs the analysis phase of **bsrsm2()**, a new sparse triangular linear system **op(A)*op(X) = α op(B)**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$\text{op}(X) = \begin{cases} X & \text{if transX} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T & \text{if transX} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ X^H & \text{if transX} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

and **op(B)** and **op(X)** are equal.

The block of BSR format is of size **blockDim*blockDim**, stored in column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires the buffer size returned by **bsrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsm2_analysis()** reports a structural zero and computes the level information stored in opaque structure **info**. The level information can extract more parallelism during a triangular solver. However **bsrsm2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrsm2_analysis()** always reports the first structural zero, even if the parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. Besides, no structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if block **A(j,j)** is missing for some **j**. The user must call **cusparseXbsrsm2_query_zero_pivot()** to know where the structural zero is.

Even when **bsrsm2_analysis()** reports a structural zero, the user can still call asynchronously **bsrsm2_solve()**. In this case, the solve will return a numerical zero in the same position as the structural zero but this result **X** is meaningless.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transX	the operation **op(B)** and **op(X)**.

mb	number of block rows of matrix **A**.
n	number of columns of matrix **op(B)** and **op(X)**.
nnzb	number of non-zero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**; larger than zero.
info	structure initialized using **cusparseCreateBsrsm2Info**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is return by **bsrsm2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	Either invalid parameters were passed (**mb, n, nnzb<=0**). Either invalid or unsupported operations were passed (**op(B) and op(X)** are different from CUSPARSE_OPERATION_NON_TRANSPOSE or CUSPARSE_OPERATION_TRANSPOSE).
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.12. cusparse<t>bsrsm2_solve()

```
cusparseStatus_t
cusparseSbsrsm2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      cusparseOperation_t transX,
                      int mb,
                      int n,
                      int nnzb,
                      const float *alpha,
                      const cusparseMatDescr_t descrA,
                      const float *bsrSortedVal,
                      const int *bsrSortedRowPtr,
                      const int *bsrSortedColInd,
                      int blockDim,
                      bsrsm2Info_t info,
                      const float *B,
                      int ldb,
                      float *X,
                      int ldx,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer)

cusparseStatus_t
cusparseDbsrsm2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      cusparseOperation_t transX,
                      int mb,
                      int n,
                      int nnzb,
                      const double *alpha,
                      const cusparseMatDescr_t descrA,
                      const double *bsrSortedVal,
                      const int *bsrSortedRowPtr,
                      const int *bsrSortedColInd,
                      int blockDim,
                      bsrsm2Info_t info,
                      const double *B,
                      int ldb,
                      double *X,
                      int ldx,
                      cusparseSolvePolicy_t policy,
                      void *pBuffer)

cusparseStatus_t
cusparseCbsrsm2_solve(cusparseHandle_t handle,
                      cusparseDirection_t dirA,
                      cusparseOperation_t transA,
                      cusparseOperation_t transX,
                      int mb,
                      int n,
                      int nnzb,
                      const cuComplex *alpha,
                      const cusparseMatDescr_t descrA,
                      const cuComplex *bsrSortedVal,
                      const int *bsrSortedRowPtr,
                      const int *bsrSortedColInd,
                      int blockDim,
                      bsrsm2Info_t info,
                      const cuComplex *B,
                      int ldb,
                      cuComplex *X,
                      int ldx,
```

This function performs the solve phase of the solution of a sparse triangular linear system:

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

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar, and

$$\text{op}(A) = \begin{cases} A & \text{if transA} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$\text{op}(X) = \begin{cases} X & \text{if transX} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T & \text{if transX} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ X^H & \text{not supported} \end{cases}$$

Only **op(A)=A** is supported.

op(B) and **op(X)** must be performed in the same way. In other words, if **op(B)=B**, **op(X)=X**.

The block of BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrsm02_solve()** can support an arbitrary **blockDim**.

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires the buffer size returned by **bsrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrsm2_solve()** can be done without level information, the user still needs to be aware of consistency. If **bsrsm2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrsm2_solve()** can be run with or without levels. On the other hand, if **bsrsm2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrsm2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsm02_solve()** has the same behavior as **bsrsv02_solve()**, reporting the first numerical zero, including a structural zero. The user must call **cusparseXbsrsm2_query_zero_pivot()** to know where the numerical zero is.

The motivation of **transpose(X)** is to improve the memory access of matrix **X**. The computational pattern of **transpose(X)** with matrix **X** in column-major order is equivalent to **X** with matrix **X** in row-major order.

In-place is supported and requires that **B** and **X** point to the same memory block, and **ldb=ldx**.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transX	the operation **op(B)** and **op(X)**.
mb	number of block rows of matrix **A**.
n	number of columns of matrix **op(B)** and **op(X)**.
nnzb	number of non-zero blocks of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) non-zero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**; larger than zero.
info	structure initialized using **cusparseCreateBsrsm2Info()**.
B	<type> right-hand-side array.
ldb	leading dimension of **B**. If **op(B)=B**, **ldb >= (mb*blockDim)**; otherwise, **ldb >= n**.
ldx	leading dimension of **X**. If **op(X)=X**, then **ldx >= (mb*blockDim)**. otherwise **ldx >= n**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **bsrsm2_bufferSize()**.

Output

X	<type> solution array with leading dimensions **ldx**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_MAPPING_ERROR	the texture binding failed.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

8.13. cusparseXbsrsm2_zeroPivot()

```
cusparseStatus_t
cusparseXbsrsm2_zeroPivot(cusparseHandle_t handle,
                          bsrsm2Info_t info,
                          int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** is either a structural zero or a numerical zero (singular block). Otherwise **position=-1**.

The **position** can be 0-base or 1-base, the same as the matrix.

Function **cusparseXbsrsm2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains a structural zero or a numerical zero if the user already called **bsrsm2_analysis()** or **bsrsm2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

8.14. cusparse<t>gemmi()

```
cusparseStatus_t
cusparseSgemmi(cusparseHandle_t handle,
               int m,
               int n,
               int k,
               int nnz,
               const float *alpha,
               const float *A,
               int lda,
               const float *cscValB,
               const int *cscColPtrB,
               const int *cscRowIndB,
               const float *beta,
               float *C,
               int ldc);

cusparseStatus_t
cusparseDgemmi(cusparseHandle_t handle,
               int m,
               int n,
               int k,
               int nnz,
               const double *alpha,
               const double *A,
               int lda,
               const double *cscValB,
               const int *cscColPtrB,
               const int *cscRowIndB,
               const double *beta,
               double *C,
               int ldc);

cusparseStatus_t
cusparseCgemmi(cusparseHandle_t handle,
               int m,
               int n,
               int k,
               int nnz,
               const cuComplex *alpha,
               const cuComplex *A,
               int lda,
               const cuComplex *cscValB,
               const int *cscColPtrB,
               const int *cscRowIndB,
               const cuComplex *beta,
               cuComplex *C,
               int ldc);

cusparseStatus_t
cusparseZgemmi(cusparseHandle_t handle,
               int m,
               int n,
               int k,
               int nnz,
               const cuDoubleComplex *alpha,
               const cuDoubleComplex *A,
               int lda,
               const cuDoubleComplex *cscValB,
               const int *cscColPtrB,
               const int *cscRowIndB,
               const cuDoubleComplex *beta,
               cuDoubleComplex *C,
```

This function performs the following matrix-matrix operations:

$$C = \alpha * A * B + \beta * C$$

A and **C** are dense matrices; **B** is a **k×n** sparse matrix that is defined in CSC storage format by the three arrays **cscValB**, **cscColPtrB**, and **cscRowIndB**); α and β are scalars; and

Remark: **B** is base-0.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrices **B** and **C**.
k	number of columns of matrix **A**.
nnz	number of nonzero elements of sparse matrix **B**.
alpha	<type> scalar used for multiplication.
A	array of dimensions **(lda, k)**.
lda	leading dimension of **A**. It must be at least **m**
cscValB	<type> array of **nnz** (= **cscColPtrB(k) – cscColPtrB(0)**) nonzero elements of matrix **B**.
cscColPtrB	integer array of **k** + 1 elements that contains the start of every row and the end of the last row plus one.
cscRowIndB	integer array of **nnz** (= **cscColPtrB(k) – cscColPtrB(0)**) column indices of the nonzero elements of matrix **B**.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least **m**

Output

C	<type> updated array of dimensions **(ldc, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n, k, nnz<0** or **lda** and **ldc** are incorrect).
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

Chapter 9. CUSPARSE EXTRA FUNCTION REFERENCE

This chapter describes the extra routines used to manipulate sparse matrices.

9.1. cusparse<t>csrgeam()

```
cusparseStatus_t
cusparseXcsrgeamNnz(cusparseHandle_t handle,
                    int m,
                    int n,
                    const cusparseMatDescr_t descrA,
                    int nnzA,
                    const int *csrRowPtrA,
                    const int *csrColIndA,
                    const cusparseMatDescr_t descrB,
                    int nnzB,
                    const int *csrRowPtrB,
                    const int *csrColIndB,
                    const cusparseMatDescr_t descrC,
                    int *csrRowPtrC,
                    int *nnzTotalDevHostPtr)
cusparseStatus_t
cusparseScsrgeam(cusparseHandle_t handle,
                 int m,
                 int n,
                 const float *alpha,
                 const cusparseMatDescr_t descrA,
                 int nnzA,
                 const float *csrValA,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 const float *beta,
                 const cusparseMatDescr_t descrB,
                 int nnzB,
                 const float *csrValB,
                 const int *csrRowPtrB,
                 const int *csrColIndB,
                 const cusparseMatDescr_t descrC,
                 float *csrValC,
                 int *csrRowPtrC,
                 int *csrColIndC)
cusparseStatus_t
cusparseDcsrgeam(cusparseHandle_t handle,
                 int m,
                 int n,
                 const double *alpha,
                 const cusparseMatDescr_t descrA,
                 int nnzA,
                 const double *csrValA,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 const double *beta,
                 const cusparseMatDescr_t descrB,
                 int nnzB,
                 const double *csrValB,
                 const int *csrRowPtrB,
                 const int *csrColIndB,
                 const cusparseMatDescr_t descrC,
                 double *csrValC,
                 int *csrRowPtrC,
                 int *csrColIndC)
cusparseStatus_t
cusparseCcsrgeam(cusparseHandle_t handle,
                 int m,
                 int n,
                 const cuComplex *alpha,
                 const cusparseMatDescr_t descrA,
                 int nnzA,
                 const cuComplex *csrValA,
```

This function performs following matrix-matrix operation

$$C = \alpha * A + \beta * B$$

where **A**, **B**, and **C** are **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrC**, and **csrColIndA|csrColIndB|csrcolIndC** respectively), and α and β are scalars. Since **A** and **B** have different sparsity patterns, cuSPARSE adopts a two-step approach to complete sparse matrix **C**. In the first step, the user allocates **csrRowPtrC** of **m +1**elements and uses function **cusparseXcsrgeamNnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC, csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgeam()** to complete matrix **C**.

The general procedure is as follows:

```
int baseC, nnzC;
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzC;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgeamNnz(handle, m, n,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrC, csrRowPtrC, nnzTotalDevHostPtr);
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(float)*nnzC);
cusparseScsrgeam(handle, m, n,
        alpha,
        descrA, nnzA,
        csrValA, csrRowPtrA, csrColIndA,
        beta,
        descrB, nnzB,
        csrValB, csrRowPtrB, csrColIndB,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

Several comments on **csrgeam()**:

- The other three combinations, NT, TN, and TT, are not supported by cuSPARSE. In order to do any one of the three, the user should use the routine **csr2csc()** to convert $A|B$ to $A^T|B^T$.
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, then the user must extend the matrix to a full one and reconfigure the **MatrixType** field of the descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.
- If the sparsity pattern of matrix **C** is known, the user can skip the call to function **cusparseXcsrgeamNnz()**. For example, suppose that the user has an iterative algorithm which would update **A** and **B** iteratively but keep the sparsity patterns.

The user can call function **cusparseXcsrgeamNnz()** once to set up the sparsity pattern of **C**, then call function **cusparse[S|D|C|Z]geam()** only for each iteration.

- The pointers **alpha** and **beta** must be valid.
- When **alpha** or **beta** is zero, it is not considered a special case by cuSPARSE. The sparsity pattern of **C** is independent of the value of **alpha** and **beta**. If the user wants $C = 0 \times A + 1 \times B^T$, then **csr2csc()** is better than **csrgeam()**.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of sparse matrix **A,B,C**.
n	number of columns of sparse matrix **A,B,C**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.
csrValB	<type> array of **nnzB** (= **csrRowPtrB(m)** - **csrRowPtrB(0)**) nonzero elements of matrix **B**.
csrRowPtrB	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndB	integer array of **nnzB** (= **csrRowPtrB(m)** - **csrRowPtrB(0)**) column indices of the nonzero elements of matrix **B**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.

Output

csrValC	<type> array of **nnzC** (= **csrRowPtrC(m)** - **csrRowPtrC(0)**) nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.

csrColIndC	integer array of **nnzC** (= **csrRowPtrC(m) - csrRowPtrC(0)**) column indices of the nonzero elements of matrixc.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m)- csrRowPtrC(0))**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**, **IndexBase** of **descrA,descrB,descrC** is not base-0 or base-1, or **alpha** or **beta** is nil)).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

9.2. cusparse<t>csrgeam2()

```
cusparseStatus_t
cusparseScsrgeam2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *alpha,
    const cusparseMatDescr_t descrA,
    int nnzA,
    const float *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const float *beta,
    const cusparseMatDescr_t descrB,
    int nnzB,
    const float *csrSortedValB,
    const int *csrSortedRowPtrB,
    const int *csrSortedColIndB,
    const cusparseMatDescr_t descrC,
    const float *csrSortedValC,
    const int *csrSortedRowPtrC,
    const int *csrSortedColIndC,
    size_t *pBufferSizeInBytes );

cusparseStatus_t
cusparseDcsrgeam2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *alpha,
    const cusparseMatDescr_t descrA,
    int nnzA,
    const double *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const double *beta,
    const cusparseMatDescr_t descrB,
    int nnzB,
    const double *csrSortedValB,
    const int *csrSortedRowPtrB,
    const int *csrSortedColIndB,
    const cusparseMatDescr_t descrC,
    const double *csrSortedValC,
    const int *csrSortedRowPtrC,
    const int *csrSortedColIndC,
    size_t *pBufferSizeInBytes );

cusparseStatus_t
cusparseCcsrgeam2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex *alpha,
    const cusparseMatDescr_t descrA,
    int nnzA,
    const cuComplex *csrSortedValA,
    const int *csrSortedRowPtrA,
    const int *csrSortedColIndA,
    const cuComplex *beta,
    const cusparseMatDescr_t descrB,
    int nnzB,
    const cuComplex *csrSortedValB,
    const int *csrSortedRowPtrB,
```

This function performs following matrix-matrix operation

$$C = \alpha * A + \beta * B$$

where **A**, **B**, and **C** are **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrC**, and **csrColIndA|csrColIndB|csrcolIndC** respectively), and α and β are scalars. Since **A** and **B** have different sparsity patterns, cuSPARSE adopts a two-step approach to complete sparse matrix **C**. In the first step, the user allocates **csrRowPtrC** of **m +1**elements and uses function **cusparseXcsrgeam2Nnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC, csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgeam2()** to complete matrix **C**.

The general procedure is as follows:

```
int baseC, nnzC;
/* alpha, nnzTotalDevHostPtr points to host memory */
size_t BufferSizeInBytes;
char *buffer = NULL;
int *nnzTotalDevHostPtr = &nnzC;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
/* prepare buffer */
cusparseScsrgeam2_bufferSizeExt(handle, m, n,
    alpha,
    descrA, nnzA,
    csrValA, csrRowPtrA, csrColIndA,
    beta,
    descrB, nnzB,
    csrValB, csrRowPtrB, csrColIndB,
    descrC,
    csrValC, csrRowPtrC, csrColIndC
    &bufferSizeInBytes
    );
cudaMalloc((void**)&buffer, sizeof(char)*bufferSizeInBytes);
cusparseXcsrgeam2Nnz(handle, m, n,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrC, csrRowPtrC, nnzTotalDevHostPtr,
        buffer);
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(float)*nnzC);
cusparseScsrgeam2(handle, m, n,
        alpha,
        descrA, nnzA,
        csrValA, csrRowPtrA, csrColIndA,
        beta,
        descrB, nnzB,
        csrValB, csrRowPtrB, csrColIndB,
        descrC,
        csrValC, csrRowPtrC, csrColIndC
        buffer);
```

Several comments on **csrgeam2()**:

- The other three combinations, NT, TN, and TT, are not supported by cuSPARSE. In order to do any one of the three, the user should use the routine **csr2csc()** to convert $A|B$ to $A^T|B^T$.
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, then the user must extend the matrix to a full one and reconfigure the **MatrixType** field of the descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.
- If the sparsity pattern of matrix **C** is known, the user can skip the call to function **cusparseXcsrgeam2Nnz()**. For example, suppose that the user has an iterative algorithm which would update **A** and **B** iteratively but keep the sparsity patterns. The user can call function **cusparseXcsrgeam2Nnz()** once to set up the sparsity pattern of **C**, then call function **cusparse[S|D|C|Z]geam()** only for each iteration.
- The pointers **alpha** and **beta** must be valid.
- When **alpha** or **beta** is zero, it is not considered a special case by cuSPARSE. The sparsity pattern of **C** is independent of the value of **alpha** and **beta**. If the user wants $C = 0 \times A + 1 \times B^T$, then **csr2csc()** is better than **csrgeam2()**.
- **csrgeam2()** is the same as **csrgeam()** except **csrgeam2()** needs explicit buffer where **csrgeam()** allocates the buffer internally.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of sparse matrix **A,B,C**.
n	number of columns of sparse matrix **A,B,C**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.
csrValB	<type> array of **nnzB** (= **csrRowPtrB(m)** - **csrRowPtrB(0)**) nonzero elements of matrix **B**.

csrRowPtrB	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndB	integer array of **nnzB** (= **csrRowPtrB(m)** - **csrRowPtrB(0)**) column indices of the nonzero elements of matrix **B**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.

Output

csrValC	<type> array of **nnzC** (= **csrRowPtrC(m)** - **csrRowPtrC(0)**) nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndC	integer array of **nnzC** (= **csrRowPtrC(m)** - **csrRowPtrC(0)**) column indices of the nonzero elements of matrixc.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m)-csrRowPtrC(0))**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**, **IndexBase** of **descrA,descrB,descrC** is not base-0 or base-1, or **alpha** or **beta** is nil)).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

9.3. cusparse<t>csrgemm()

```
cusparseStatus_t
cusparseXcsrgemmNnz(cusparseHandle_t handle,
                    cusparseOperation_t transA,
                    cusparseOperation_t transB,
                    int m,
                    int n,
                    int k,
                    const cusparseMatDescr_t descrA,
                    const int nnzA,
                    const int *csrRowPtrA,
                    const int *csrColIndA,
                    const cusparseMatDescr_t descrB,
                    const int nnzB,
                    const int *csrRowPtrB,
                    const int *csrColIndB,
                    const cusparseMatDescr_t descrC,
                    int *csrRowPtrC,
                    int *nnzTotalDevHostPtr )
cusparseStatus_t
cusparseScsrgemm(cusparseHandle_t handle,
                 cusparseOperation_t transA,
                 cusparseOperation_t transB,
                 int m,
                 int n,
                 int k,
                 const cusparseMatDescr_t descrA,
                 const int nnzA,
                 const float *csrValA,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 const cusparseMatDescr_t descrB,
                 const int nnzB,
                 const float *csrValB,
                 const int *csrRowPtrB,
                 const int *csrColIndB,
                 const cusparseMatDescr_t descrC,
                 float *csrValC,
                 const int *csrRowPtrC,
                 int *csrColIndC )
cusparseStatus_t
cusparseDcsrgemm(cusparseHandle_t handle,
                 cusparseOperation_t transA,
                 cusparseOperation_t transB,
                 int m,
                 int n,
                 int k,
                 const cusparseMatDescr_t descrA,
                 const int nnzA,
                 const double *csrValA,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 const cusparseMatDescr_t descrB,
                 const int nnzB,
                 const double *csrValB,
                 const int *csrRowPtrB,
                 const int *csrColIndB,
                 const cusparseMatDescr_t descrC,
                 double *csrValC,
                 const int *csrRowPtrC,
                 int *csrColIndC )
cusparseStatus_t
cusparseCcsrgemm(cusparseHandle_t handle,
                 cusparseOperation_t transA,
```

This function performs following matrix-matrix operation:

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

where $\mathrm{op}(A)$, $\mathrm{op}(B)$ and **C** are **m×k**, **k×n**, and **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrC**, and **csrColIndA|csrColIndB|csrcolIndC** respectively. The operation is defined by

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans != CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \end{cases}$$

There are four versions, NN, NT, TN, and TT. NN stands for $C = A * B$, NT stands for $C = A * B^T$, TN stands for $C = A^T * B$ and TT stands for $C = A^T * B^T$.

The cuSPARSE library adopts a two-step approach to complete sparse matrix. In the first step, the user allocates **csrRowPtrC** of **m+1** elements and uses the function **cusparseXcsrgemmNnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (the number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC** and **csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgemm()** to complete matrix **C**.

The general procedure is as follows:

```
int baseC, nnzC;
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzC;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgemmNnz(handle, transA, transB, m, n, k,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrC, csrRowPtrC, nnzTotalDevHostPtr );
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(float)*nnzC);
cusparseScsrgemm(handle, transA, transB, m, n, k,
        descrA, nnzA,
        csrValA, csrRowPtrA, csrColIndA,
        descrB, nnzB,
        csrValB, csrRowPtrB, csrColIndB,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

Several comments on **csrgemm()**:

- Although NN, NT, TN and TT are supported, only the NN version is implemented. For the NT, TN and TT versions, **csr2csc()** is used to transpose the relevant matrices, followed by a call to the NN version of **csrgemm()**.
- The NN version needs working space of size **nnzA** integers at least.

- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, the user must extend the matrix to a full one and reconfigure the **MatrixType** field descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.
- Only devices of compute capability 2.0 or above are supported.

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$
transB	the operation $\mathrm{op}(B)$
m	number of rows of sparse matrix $\mathrm{op}(A)$ and **C**.
n	number of columns of sparse matrix $\mathrm{op}(B)$ and **C**.
k	number of columns/rows of sparse matrix $\mathrm{op}(A)$ / $\mathrm{op}(B)$.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of $\tilde{m}+1$ elements that contains the start of every row and the end of the last row plus one. $\tilde{m}=m$ if **transA** == **CUSPARSE_OPERATION_NON_TRANSPOSE**, otherwise $\tilde{m}=k$.
csrColIndA	integer array of **nnzA** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.
csrValB	<type> array of **nnzB** nonzero elements of matrix **B**.
csrRowPtrB	integer array of $\tilde{k}+1$ elements that contains the start of every row and the end of the last row plus one. $\tilde{k}=k$ if **transB** == **CUSPARSE_OPERATION_NON_TRANSPOSE**, otherwise $\tilde{k}=n$
csrColIndB	integer array of **nnzB** column indices of the nonzero elements of matrix **B**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.

Output

csrValC	<type> array of **nnzC** (= **csrRowPtrC(m)** - **csrRowPtrC(0)**) nonzero elements of matrix **C**.

csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	integer array of **nnzC** (= **csrRowPtrC(m)** - **csrRowPtrC(0)**) column indices of the nonzero elements of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m)- csrRowPtrC(0))**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,k<0**; **IndexBase** of **descrA,descrB,descrC** is not base-0 or base-1; or **alpha** or **beta** is nil)).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

9.4. cusparse<t>csrgemm2()

```
cusparseStatus_t
cusparseScsrgemm2_bufferSizeExt(cusparseHandle_t handle,
                                int m,
                                int n,
                                int k,
                                const float *alpha,
                                const cusparseMatDescr_t descrA,
                                int nnzA,
                                const int *csrRowPtrA,
                                const int *csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int nnzB,
                                const int *csrRowPtrB,
                                const int *csrColIndB,
                                const float *beta,
                                const cusparseMatDescr_t descrD,
                                int nnzD,
                                const int *csrRowPtrD,
                                const int *csrColIndD,
                                csrgemm2Info_t info,
                                size_t *pBufferSizeInBytes );

cusparseStatus_t
cusparseDcsrgemm2_bufferSizeExt(cusparseHandle_t handle,
                                int m,
                                int n,
                                int k,
                                const double *alpha,
                                const cusparseMatDescr_t descrA,
                                int nnzA,
                                const int *csrRowPtrA,
                                const int *csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int nnzB,
                                const int *csrRowPtrB,
                                const int *csrColIndB,
                                const double *beta,
                                const cusparseMatDescr_t descrD,
                                int nnzD,
                                const int *csrRowPtrD,
                                const int *csrColIndD,
                                csrgemm2Info_t info,
                                size_t *pBufferSizeInBytes );

cusparseStatus_t
cusparseCcsrgemm2_bufferSizeExt(cusparseHandle_t handle,
                                int m,
                                int n,
                                int k,
                                const cuComplex *alpha,
                                const cusparseMatDescr_t descrA,
                                int nnzA,
                                const int *csrRowPtrA,
                                const int *csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int nnzB,
                                const int *csrRowPtrB,
                                const int *csrColIndB,
                                const cuComplex *beta,
                                const cusparseMatDescr_t descrD,
                                int nnzD,
                                const int *csrRowPtrD,
                                const int *csrColIndD,
```

This function performs following matrix-matrix operation:

$$C = \text{alpha} * A * B + \text{beta} * D$$

where **A**, **B**, **D** and **C** are **m×k**, **k×n**, **m×n** and **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValD|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrD|csrRowPtrC**, and **csrColIndA|csrColIndB|csrColIndD|csrcolIndC** respectively.

We provide **csrgemm2** as a generalization of **csrgemm**. It provides more operations in terms of **alpha** and **beta**. For example, **C = -A*B+D** can be done by **csrgemm2**.

The **csrgemm2** uses **alpha** and **beta** to support the following operations:

alpha	beta	operation
NULL	NULL	invalid
NULL	!NULL	C = beta*D, A and B are not used
!NULL	NULL	C = alpha*A*B, D is not used
!NULL	!NULL	C = alpha*A*B + beta*D

The numerical value of **alpha** and **beta** only affects the numerical values of **C**, not its sparsity pattern. For example, if **alpha** and **beta** are not zero, the sparsity pattern of **C** is union of **A*B** and **D**, independent of numerical value of **alpha** and **beta**.

The following table shows different operations according to the value of **m**, **n** and **k**

m,n,k	operation
m<0 or n <0 or k<0	invalid
m is 0 or n is 0	do nothing
m >0 and n >0 and k is 0	invalid if beta is zero; C = beta*D if beta is not zero.
m >0 and n >0 and k >0	C = beta*D if alpha is zero. C = alpha*A*B if beta is zero. C = alpha*A*B + beta*D if alpha and beta are not zero.

This function requires the buffer size returned by **csrgemm2_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The cuSPARSE library adopts a two-step approach to complete sparse matrix. In the first step, the user allocates **csrRowPtrC** of **m+1** elements and uses the function **cusparseXcsrgemm2Nnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (the number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC** and **csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgemm2()** to evaluate matrix **C**.

The general procedure of **C=-A*B+D** is as follows:

```
// assume matrices A, B and D are ready.
int baseC, nnzC;
csrgemm2Info_t info = NULL;
size_t bufferSize;
void *buffer = NULL;
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzC;
double alpha = -1.0;
double beta  =  1.0;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);

// step 1: create an opaque structure
cusparseCreateCsrgemm2Info(&info);

// step 2: allocate buffer for csrgemm2Nnz and csrgemm2
cusparseDcsrgemm2_bufferSizeExt(handle, m, n, k, &alpha,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    &beta,
    descrD, nnzD, csrRowPtrD, csrColIndD,
    info,
    &bufferSize);
cudaMalloc(&buffer, bufferSize);

// step 3: compute csrRowPtrC
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgemm2Nnz(handle, m, n, k,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrD, nnzD, csrRowPtrD, csrColIndD,
        descrC, csrRowPtrC, nnzTotalDevHostPtr,
        info, buffer );
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}

// step 4: finish sparsity pattern and value of C
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(double)*nnzC);
// Remark: set csrValC to null if only sparsity pattern is required.
cusparseDcsrgemm2(handle, m, n, k, &alpha,
        descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrValB, csrRowPtrB, csrColIndB,
        &beta,
        descrD, nnzD, csrValD, csrRowPtrD, csrColIndD,
        descrC, csrValC, csrRowPtrC, csrColIndC,
        info, buffer);

// step 5: destroy the opaque structure
cusparseDestroyCsrgemm2Info(info);
```

Several comments on **csrgemm2()**:

- Only the NN version is supported. For other modes, the user has to transpose **A** or **B** explicitly.
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, the user must extend the matrix to a full one and reconfigure the **MatrixType** field descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.

- if **csrValC** is zero, only sparisty pattern of **C** is calculated.
- Only devices of compute capability 2.0 or above are supported.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of sparse matrix **A**, **D** and **C**.
n	number of columns of sparse matrix **B**, **D** and **C**.
k	number of columns/rows of sparse matrix **A** / **B**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnzA** column indices of the nonzero elements of matrix **A**.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.
csrValB	<type> array of **nnzB** nonzero elements of matrix **B**.
csrRowPtrB	integer array of **k+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndB	integer array of **nnzB** column indices of the nonzero elements of matrix **B**.
descrD	the descriptor of matrix **D**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzD	number of nonzero elements of sparse matrix **D**.
csrValD	<type> array of **nnzD** nonzero elements of matrix **D**.
csrRowPtrD	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndD	integer array of **nnzD** column indices of the nonzero elements of matrix **D**.
beta	<type> scalar used for multiplication.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
info	structure with information used in **csrgemm2Nnz** and **csrgemm2**.

pBuffer	buffer allocated by the user; the size is returned by **csrgemm2_bufferSizeExt**.

Output

csrValC	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	integer array of **nnzC** column indices of the nonzero elements of matrix **C**.
pBufferSizeInBytes	number of bytes of the buffer used in **csrgemm2Nnnz** and **csrgemm2**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m) - csrRowPtrC(0))**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,k<0**; **IndexBase** of **descrA,descrB,descrD,descrC** is not base-0 or base-1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

Chapter 10. CUSPARSE PRECONDITIONERS REFERENCE

This chapter describes the routines that implement different preconditioners.

In particular, the incomplete factorizations are implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate **`csrsv_analysis()`** function. The analysis is specific to the sparsity pattern of the given matrix and the selected **`cusparseOperation_t`** type. The information from the analysis phase is stored in the parameter of type **`cusparseSolveAnalysisInfo_t`** that has been initialized previously with a call to **`cusparseCreateSolveAnalysisInfo()`**.

Second, during the numerical factorization phase, the given coefficient matrix is factorized using the information stored in the **`cusparseSolveAnalysisInfo_t`** parameter by calling the appropriate **`csrilu0()`** or **`csric0()`** function.

The analysis phase is shared across the sparse triangular solve, and the incomplete factorization and must be performed only once. The resulting information can be passed to the numerical factorization and the sparse triangular solve multiple times.

Finally, once the incomplete factorization and all the sparse triangular solves have completed, the opaque data structure pointed to by the **`cusparseSolveAnalysisInfo_t`** parameter can be released by calling **`cusparseDestroySolveAnalysisInfo()`**.

10.1. Incomplete Cholesky Factorization: level 0

Different algorithms for ic0 are discussed in this section.

10.1.1. cusparse<t>csric0()

```
cusparseStatus_t
cusparseScsric0(cusparseHandle_t handle,
                cusparseOperation_t trans,
                int m,
                const cusparseMatDescr_t descrA,
                float *csrValM,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseDcsric0(cusparseHandle_t handle,
                cusparseOperation_t trans,
                int m,
                const cusparseMatDescr_t descrA,
                double *csrValM,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseCcsric0(cusparseHandle_t handle,
                cusparseOperation_t trans,
                int m,
                const cusparseMatDescr_t descrA,
                cuComplex *csrValM,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseZcsric0(cusparseHandle_t handle,
                cusparseOperation_t trans,
                int m,
                const cusparseMatDescr_t descrA,
                cuDoubleComplex *csrValM,
                const int *csrRowPtrA,
                const int *csrColIndA,
                cusparseSolveAnalysisInfo_t info)
```

This function computes the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$op(A) \approx R^T R$$

A is an **m** × **m** Hermitian/symmetric positive definite sparse matrix that is defined in CSR storage format by the three arrays **csrValM**, **csrRowPtrA**, and **csrColIndA**; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Notice that only a lower or upper Hermitian/symmetric part of the matrix **A** is actually stored. It is overwritten by the lower or upper triangular factors R^T and R, respectively.

A call to this routine must be preceded by a call to the **csrsv_analysis()** routine.

The matrix descriptor for **csrsv_analysis()** and **csric0()** must be the same. Otherwise, runtime error would occur.

This function requires some extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation **op** (A).
m	number of rows and columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix types are **CUSPARSE_MATRIX_TYPE_SYMMETRIC** and **CUSPARSE_MATRIX_TYPE_HERMITIAN**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValM	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).

Output

csrValM	<type> matrix containing the incomplete-Cholesky lower or upper triangular factor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.2. cusparse<t>csric02_bufferSize()

```
cusparseStatus_t
cusparseScsric02_bufferSize(cusparseHandle_t handle,
                            int m,
                            int nnz,
                            const cusparseMatDescr_t descrA,
                            float *csrValA,
                            const int *csrRowPtrA,
                            const int *csrColIndA,
                            csric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDcsric02_bufferSize(cusparseHandle_t handle,
                            int m,
                            int nnz,
                            const cusparseMatDescr_t descrA,
                            double *csrValA,
                            const int *csrRowPtrA,
                            const int *csrColIndA,
                            csric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCcsric02_bufferSize(cusparseHandle_t handle,
                            int m,
                            int nnz,
                            const cusparseMatDescr_t descrA,
                            cuComplex *csrValA,
                            const int *csrRowPtrA,
                            const int *csrColIndA,
                            csric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZcsric02_bufferSize(cusparseHandle_t handle,
                            int m,
                            int nnz,
                            const cusparseMatDescr_t descrA,
                            cuDoubleComplex *csrValA,
                            const int *csrRowPtrA,
                            const int *csrColIndA,
                            csric02Info_t info,
                            int *pBufferSizeInBytes);
```

This function returns size of buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

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

The buffer size depends on dimension **m** and **nnz**, the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call **csric02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **csric02_analysis()** and **csric02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.3. cusparse<t>csric02_analysis()

```
cusparseStatus_t
cusparseScsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const float *csrValA,
                          const int *csrRowPtrA,
                          const int *csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseDcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const double *csrValA,
                          const int *csrRowPtrA,
                          const int *csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseCcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const cuComplex *csrValA,
                          const int *csrRowPtrA,
                          const int *csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseZcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex *csrValA,
                          const int *csrRowPtrA,
                          const int *csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);
```

This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

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

This function requires a buffer size returned by **csric02_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csric02_analysis()** reports a structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete Cholesky factorization. However **csric02()** can be done without level information. To disable level information, the user must specify the policy of **csric02_analysis()** and **csric02()** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **csric02_analysis()** always reports the first structural zero, even if the policy is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. The user needs to call **cusparseXcsric02_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **csric02()** if **csric02_analysis()** reports a structural zero. In this case, the user can still call **csric02()**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure initialized using **cusparseCreateCsric02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **csric02_bufferSize()**.

Output

info	number of bytes of the buffer used in **csric02_analysis()** and **csric02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.4. cusparse<t>csric02()

```
cusparseStatus_t
cusparseScsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 float *csrValA_valM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseDcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 double *csrValA_valM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseCcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 cuComplex *csrValA_valM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseZcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex *csrValA_valM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);
```

This function performs the solve phase of the computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

This function requires a buffer size returned by **csric02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **csric02()** can be done without level information, the user still needs to be aware of consistency. If **csric02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csric02()** can be run with or without levels. On the other hand, if **csric02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csric02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csric02()** reports the first numerical zero, including a structural zero. The user must call **cusparseXcsric02_zeroPivot()** to know where the numerical zero is.

Function **csric02()** only takes the lower triangular part of matrix **A** to perform factorization. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, the fill mode and diagonal type are ignored, and the strictly upper triangular part is ignored and never touched. It does not matter if **A** is Hermitian or not. In other words, from the point of view of **csric02()** **A** is Hermitian and only the lower triangular part is provided.

In practice, a positive definite matrix may not have incomplete cholesky factorization. To the best of our knowledge, only matrix **M** can guarantee the existence of incomplete cholesky factorization. If **csric02()** failed cholesky factorization and reported a numerical zero, it is possible that incomplete cholesky factorization does not exist.

For example, suppose **A** is a real **m** × **m** matrix, the following code solves the precondition system **M*y = x** where **M** is the product of Cholesky factorization **L** and its transpose.

$$M = LL^H$$

```
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
csric02Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for csric02 and two info's for csrsv2
cusparseCreateCsric02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_Lt);

// step 3: query how much memory used in csric02 and csrsv2, and allocate the
 buffer
cusparseDcsric02_bufferSize(handle, m, nnz,
    descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &bufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_Lt, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_Lt,&pBufferSize_Lt);

pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// we can do analysis of csric02 and csrsv2 simultaneously.

cusparseDcsric02_analysis(handle, m, nnz, descr_M,
    d_csrVal, d_csrRowPtr, d_csrColInd, info_M,
    policy_M, pBuffer);
status = cusparseXcsric02_zeroPivot(handle, info_M, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
```

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA_valM	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **csric02_bufferSize()**.

Output

csrValA_valM	<type> matrix containing the incomplete-Cholesky lower triangular factor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.5. cusparseXcsric02_zeroPivot()

```
cusparseStatus_t
cusparseXcsric02_zeroPivot(cusparseHandle_t handle,
                           csric02Info_t info,
                           int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero; otherwise, **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsric02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csric02_analysis()** or **csric02()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **L(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.1.6. cusparse<t>bsric02_bufferSize()

```
cusparseStatus_t
cusparseSbsric02_bufferSize(cusparseHandle_t handle,
                            cusparseDirection_t dirA,
                            int mb,
                            int nnzb,
                            const cusparseMatDescr_t descrA,
                            float *bsrValA,
                            const int *bsrRowPtrA,
                            const int *bsrColIndA,
                            int blockDim,
                            bsric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDbsric02_bufferSize(cusparseHandle_t handle,
                            cusparseDirection_t dirA,
                            int mb,
                            int nnzb,
                            const cusparseMatDescr_t descrA,
                            double *bsrValA,
                            const int *bsrRowPtrA,
                            const int *bsrColIndA,
                            int blockDim,
                            bsric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCbsric02_bufferSize(cusparseHandle_t handle,
                            cusparseDirection_t dirA,
                            int mb,
                            int nnzb,
                            const cusparseMatDescr_t descrA,
                            cuComplex *bsrValA,
                            const int *bsrRowPtrA,
                            const int *bsrColIndA,
                            int blockDim,
                            bsric02Info_t info,
                            int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZbsric02_bufferSize(cusparseHandle_t handle,
                            cusparseDirection_t dirA,
                            int mb,
                            int nnzb,
                            const cusparseMatDescr_t descrA,
                            cuDoubleComplex *bsrValA,
                            const int *bsrRowPtrA,
                            const int *bsrColIndA,
                            int blockDim,
                            bsric02Info_t info,
                            int *pBufferSizeInBytes);
```

This function returns the size of a buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)*(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**.

The buffer size depends on the dimensions of **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsric02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **bsric02_analysis()** and **bsric02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**); the base index is not 0 or 1.

CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.7. cusparse<t>bsric02_analysis()

```
cusparseStatus_t
cusparseSbsric02_analysis(cusparseHandle_t handle,
                          cusparseDirection_t dirA,
                          int mb,
                          int nnzb,
                          const cusparseMatDescr_t descrA,
                          const float *bsrValA,
                          const int *bsrRowPtrA,
                          const int *bsrColIndA,
                          int blockDim,
                          bsric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseDbsric02_analysis(cusparseHandle_t handle,
                          cusparseDirection_t dirA,
                          int mb,
                          int nnzb,
                          const cusparseMatDescr_t descrA,
                          const double *bsrValA,
                          const int *bsrRowPtrA,
                          const int *bsrColIndA,
                          int blockDim,
                          bsric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseCbsric02_analysis(cusparseHandle_t handle,
                          cusparseDirection_t dirA,
                          int mb,
                          int nnzb,
                          const cusparseMatDescr_t descrA,
                          const cuComplex *bsrValA,
                          const int *bsrRowPtrA,
                          const int *bsrColIndA,
                          int blockDim,
                          bsric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);

cusparseStatus_t
cusparseZbsric02_analysis(cusparseHandle_t handle,
                          cusparseDirection_t dirA,
                          int mb,
                          int nnzb,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex *bsrValA,
                          const int *bsrRowPtrA,
                          const int *bsrColIndA,
                          int blockDim,
                          bsric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void *pBuffer);
```

This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsric02_bufferSize90**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function**bsric02_analysis()** reports structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete Cholesky factorization. However **bsric02()** can be done without level information. To disable level information, the user needs to specify the parameter **policy** of **bsric02[_analysis|]** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsric02_analysis** always reports the first structural zero, even when parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. The user must call **cusparseXbsric02_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsric02()** if **bsric02_analysis()** reports a structural zero. In this case, the user can still call **bsric02()**, which returns a numerical zero in the same position as the structural zero. However the result is meaningless.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.

blockDim	block dimension of sparse matrix A; must be larger than zero.
info	structure initialized using **cusparseCreateBsric02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **bsric02_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**); the base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.8. cusparse<t>bsric02()

```
cusparseStatus_t
cusparseSbsric02(cusparseHandle_t handle,
                 cusparseDirection_t dirA,
                 int mb,
                 int nnzb,
                 const cusparseMatDescr_t descrA,
                 float *bsrValA,
                 const int *bsrRowPtrA,
                 const int *bsrColIndA,
                 int blockDim,
                 bsric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseDbsric02(cusparseHandle_t handle,
                 cusparseDirection_t dirA,
                 int mb,
                 int nnzb,
                 const cusparseMatDescr_t descrA,
                 double *bsrValA,
                 const int *bsrRowPtrA,
                 const int *bsrColIndA,
                 int blockDim,
                 bsric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseCbsric02(cusparseHandle_t handle,
                 cusparseDirection_t dirA,
                 int mb,
                 int nnzb,
                 const cusparseMatDescr_t descrA,
                 cuComplex *bsrValA,
                 const int *bsrRowPtrA,
                 const int *bsrColIndA,
                 int blockDim,
                 bsric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);

cusparseStatus_t
cusparseZbsric02(cusparseHandle_t handle,
                 cusparseDirection_t dirA,
                 int mb,
                 int nnzb,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex *bsrValA,
                 const int *bsrRowPtrA,
                 const int *bsrColIndA,
                 int blockDim,
                 bsric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void *pBuffer);
```

This function performs the solve phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsric02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsric02()** can be done without level information, the user must be aware of consistency. If **bsric02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsric02()** can be run with or without levels. On the other hand, if **bsric02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsric02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsric02()** has the same behavior as **csric02()**. That is, **bsr2csr(bsric02(A)) = csric02(bsr2csr(A))**. The numerical zero of **csric02()** means there exists some zero **L(j,j)**. The numerical zero of **bsric02()** means there exists some block **Lj,j)** that is not invertible.

Function **bsric02** reports the first numerical zero, including a structural zero. The user must call **cusparseXbsric02_zeroPivot()** to know where the numerical zero is.

The **bsric02()** function only takes the lower triangular part of matrix **A** to perform factorization. The strictly upper triangular part is ignored and never touched. It does not matter if **A** is Hermitian or not. In other words, from the point of view of **bsric02()**, **A** is Hermitian and only the lower triangular part is provided. Moreover, the imaginary part of diagonal elements of diagonal blocks is ignored.

For example, suppose **A** is a real m-by-m matrix, where **m=mb*blockDim**. The following code solves precondition system **M*y = x**, where **M** is the product of Cholesky factorization **L** and its transpose.

$$M = LL^H$$

```
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
bsric02Info_t info_M  = 0;
bsrsv2Info_t  info_L  = 0;
bsrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsric02 and two info's for bsrsv2
cusparseCreateBsric02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_Lt);

// step 3: query how much memory used in bsric02 and bsrsv2, and allocate the
 buffer
cusparseDbsric02_bufferSize(handle, dir, mb, nnzb,
    descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M,
 &bufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
 &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_Lt, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_Lt,
 &pBufferSize_Lt);

pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
```

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **bsric02_bufferSize()**.

Output

bsrValA	<type> matrix containing the incomplete-Cholesky lower triangular factor.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**); the base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.1.9. cusparseXbsric02_zeroPivot()

```
cusparseStatus_t
cusparseXbsric02_zeroPivot(cusparseHandle_t handle,
                           bsric02Info_t info,
                           int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero (the block is not positive definite). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsric02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains a structural zero or a numerical zero if the user already called **bsric02_analysis()** or **bsric02()**.

Output

position	if no structural or numerical zero, **position** is -1, otherwise if **A(j,j)** is missing or **L(j,j)** is not positive definite, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.2. Incomplete LU Factorization: level 0

Different algorithms for ilu0 are discussed in this section.

10.2.1. cusparse<t>csrilu0()

```
cusparseStatus_t
cusparseScsrilu0(cusparseHandle_t handle,
                 cusparseOperation_t trans,
                 int m,
                 const cusparseMatDescr_t descrA,
                 float *csrValM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseDcsrilu0(cusparseHandle_t handle,
                 cusparseOperation_t trans,
                 int m,
                 const cusparseMatDescr_t descrA,
                 double *csrValM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseCcsrilu0(cusparseHandle_t handle,
                 cusparseOperation_t trans,
                 int m,
                 const cusparseMatDescr_t descrA,
                 cuComplex *csrValM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 cusparseSolveAnalysisInfo_t info)
cusparseStatus_t
cusparseZcsrilu0(cusparseHandle_t handle,
                 cusparseOperation_t trans,
                 int m,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex *csrValM,
                 const int *csrRowPtrA,
                 const int *csrColIndA,
                 cusparseSolveAnalysisInfo_t info)
```

This function computes the incomplete-LU factorization with 0 fill-in and no pivoting:

$$op(A) \approx LU$$

A is an **m** × **m** sparse matrix that is defined in CSR storage format by the three arrays **csrValM**, **csrRowPtrA**, and **csrColIndA**; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Notice that the diagonal of lower triangular factor L is unitary and need not be stored. Therefore, the input matrix is overwritten with the resulting lower and upper triangular factors L and U, respectively.

A call to this routine must be preceded by a call to the **csrsv_analysis()** routine.

The matrix descriptor for **csrsv_analysis()** and **csrilu0()** must be the same. Otherwise, runtime error would occur.

This function requires some extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
trans	the operation **op** (A).
m	number of rows and columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValM	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).

Output

csrValM	<type> matrix containing the incomplete-LU lower and upper triangular factors.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.2. cusparseCsrilu0Ex()

```
cusparseStatus_t CUSPARSEAPI cusparseCsrilu0Ex(cusparseHandle_t handle,
                                               cusparseOperation_t
 trans,
                                               int m,
                                               const cusparseMatDescr_t
 descrA,
                                               void *csrSortedValA_ValM,
 cudaDataType csrSortedValA_ValMtype,
                                               const int
 *csrSortedRowPtrA,
                                               const int
 *csrSortedColIndA,

 cusparseSolveAnalysisInfo_t info,
                                               cudaDataType
 executiontype);
```

This function is an extended version of **cusparse<t>csrilu0()**. For detailed description of the functionality, see **cusparse<t>csrilu0()**.

This function does not support half-precision execution type, but it supports half-precision IO with single precision execution.

Input specifically required by cusparseCsrilu0Ex

csrSortedValA_ValMtype	Data type of **csrSortedValA_ValM**.
executiontype	Data type used for computation.

10.2.3. cusparse<t>csrilu02_numericBoost()

```
cusparseStatus_t
cusparseScsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               float *boost_val);

cusparseStatus_t
cusparseDcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               double *boost_val);

cusparseStatus_t
cusparseCcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               cuComplex *boost_val);

cusparseStatus_t
cusparseZcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               cuDoubleComplex *boost_val);
```

The user can use a boost value to replace a numerical value in incomplete LU factorization. The **tol** is used to determine a numerical zero, and the **boost_val** is used to replace a numerical zero. The behavior is

if **tol >= fabs(A(j,j))**, then **A(j,j)=boost_val**.

To enable a boost value, the user has to set parameter **enable_boost** to 1 before calling **csrilu02()**. To disable a boost value, the user can call **csrilu02_numericBoost()** again with parameter **enable_boost=0**.

If **enable_boost=0**, **tol** and **boost_val** are ignored.

Both **tol** and **boost_val** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	structure initialized using **cusparseCreateCsrilu02Info()**.
enable_boost	disable boost by **enable_boost=0**; otherwise, boost is enabled.
tol	tolerance to determine a numerical zero.
boost_val	boost value to replace a numerical zero.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** or pointer mode is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.2.4. cusparse<t>csrilu02_bufferSize()

```
cusparseStatus_t
cusparseScsrilu02_bufferSize(cusparseHandle_t handle,
                             int m,
                             int nnz,
                             const cusparseMatDescr_t descrA,
                             float *csrValA,
                             const int *csrRowPtrA,
                             const int *csrColIndA,
                             csrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDcsrilu02_bufferSize(cusparseHandle_t handle,
                             int m,
                             int nnz,
                             const cusparseMatDescr_t descrA,
                             double *csrValA,
                             const int *csrRowPtrA,
                             const int *csrColIndA,
                             csrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCcsrilu02_bufferSize(cusparseHandle_t handle,
                             int m,
                             int nnz,
                             const cusparseMatDescr_t descrA,
                             cuComplex *csrValA,
                             const int *csrRowPtrA,
                             const int *csrColIndA,
                             csrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZcsrilu02_bufferSize(cusparseHandle_t handle,
                             int m,
                             int nnz,
                             const cusparseMatDescr_t descrA,
                             cuDoubleComplex *csrValA,
                             const int *csrRowPtrA,
                             const int *csrColIndA,
                             csrilu02Info_t info,
                             int *pBufferSizeInBytes);
```

This function returns size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

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

The buffer size depends on the dimension **m** and **nnz**, the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call **csrilu02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **csrilu02_analysis()** and **csrilu02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.5. cusparse<t>csrilu02_analysis()

```
cusparseStatus_t
cusparseScsrilu02_analysis(cusparseHandle_t handle,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           const float *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseDcsrilu02_analysis(cusparseHandle_t handle,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           const double *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseCcsrilu02_analysis(cusparseHandle_t handle,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           const cuComplex *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseZcsrilu02_analysis(cusparseHandle_t handle,
                           int m,
                           int nnz,
                           const cusparseMatDescr_t descrA,
                           const cuDoubleComplex *csrValA,
                           const int *csrRowPtrA,
                           const int *csrColIndA,
                           csrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);
```

This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

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

This function requires the buffer size returned by **csrilu02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrilu02_analysis()** reports a structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete LU factorization; however **csrilu02()** can be done without level information. To disable level information, the user must specify the policy of **csrilu02()** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

It is the user's choice whether to call **csrilu02()** if **csrilu02_analysis()** reports a structural zero. In this case, the user can still call **csrilu02()**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure initialized using **cusparseCreateCsrilu02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrilu02_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.6. cusparse<t>csrilu02()

```
cusparseStatus_t
cusparseScsrilu02(cusparseHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  float *csrValA_valM,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  csrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseDcsrilu02(cusparseHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  double *csrValA_valM,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  csrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseCcsrilu02(cusparseHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  cuComplex *csrValA_valM,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  csrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseZcsrilu02(cusparseHandle_t handle,
                  int m,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  cuDoubleComplex *csrValA_valM,
                  const int *csrRowPtrA,
                  const int *csrColIndA,
                  csrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);
```

This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

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

This function requires a buffer size returned by **csrilu02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**. The fill mode and diagonal type are ignored.

Although **csrilu02()** can be done without level information, the user still needs to be aware of consistency. If **csrilu02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrilu02()** can be run with or without levels. On the other hand, if **csrilu02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrilu02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrilu02()** reports the first numerical zero, including a structural zero. The user must call **cusparseXcsrilu02_zeroPivot()** to know where the numerical zero is.

For example, suppose **A** is a real **m × m** matrix, the following code solves precondition system **M*y = x** where **M** is the product of LU factors **L** and **U**.

```
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
csrilu02Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_U  = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U  = CUSPARSE_OPERATION_NON_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);

cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for csrilu02 and two info's for csrsv2
cusparseCreateCsrilu02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_U);

// step 3: query how much memory used in csrilu02 and csrsv2, and allocate the
 buffer
cusparseDcsrilu02_bufferSize(handle, m, nnz,
    descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &pBufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_U, m, nnz,
    descr_U, d_csrVal, d_csrRowPtr, d_csrColInd, info_U, &pBufferSize_U);

pBufferSize = max(pBufferSize_M, max(pBufferSize_L, pBufferSize_U));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
```

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA_valM	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** + 1 elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **csrilu02_bufferSize()**.

Output

csrValA_valM	<type> matrix containing the incomplete-LU lower and upper triangular factors.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<=0**), base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.7. cusparseXcsrilu02_zeroPivot()

```
cusparseStatus_t
cusparseXcsrilu02_zeroPivot(cusparseHandle_t handle,
                            csrilu02Info_t info,
                            int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero; otherwise, **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrilu02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrilu02_analysis()** or **csrilu02()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.2.8. cusparse<t>bsrilu02_numericBoost()

```
cusparseStatus_t
cusparseSbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               float *boost_val);

cusparseStatus_t
cusparseDbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               double *boost_val);

cusparseStatus_t
cusparseCbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               cuComplex *boost_val);

cusparseStatus_t
cusparseZbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t info,
                               int enable_boost,
                               double *tol,
                               cuDoubleComplex *boost_val);
```

The user can use a boost value to replace a numerical value in incomplete LU factorization. Parameter **tol** is used to determine a numerical zero, and **boost_val** is used to replace a numerical zero. The behavior is as follows:

if **tol >= fabs(A(j,j))**, then reset each diagonal element of block **A(j,j)** by **boost_val**.

To enable a boost value, the user sets parameter **enable_boost** to 1 before calling **bsrilu02()**. To disable the boost value, the user can call **bsrilu02_numericBoost()** with parameter **enable_boost=0**.

If **enable_boost=0**, **tol** and **boost_val** are ignored.

Both **tol** and **boost_val** can be in host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	structure initialized using **cusparseCreateBsrilu02Info()**.
enable_boost	disable boost by setting **enable_boost=0**. Otherwise, boost is enabled.
tol	tolerance to determine a numerical zero.

boost_val	boost value to replace a numerical zero.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** or pointer mode is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.2.9. cusparse<t>bsrilu02_bufferSize()

```
cusparseStatus_t
cusparseSbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             float *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             double *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             cuComplex *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             cuDoubleComplex *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);
```

This function returns the size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)*(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**.

The buffer size depends on the dimensions of **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsrilu02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **bsrilu02_analysis()** and **bsrilu02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**), base index is not 0 or 1.

CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.10. cusparse<t>bsrilu02_analysis()

```
cusparseStatus_t
cusparseSbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           float *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseDbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           double *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseCbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);

cusparseStatus_t
cusparseZbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex *bsrValA,
                           const int *bsrRowPtrA,
                           const int *bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void *pBuffer);
```

This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsrilu02_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrilu02_analysis()** reports a structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete LU factorization. However **bsrilu02()** can be done without level information. To disable level information, the user needs to specify the parameter **policy** of **bsrilu02[_analysis|]** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrilu02_analysis()** always reports the first structural zero, even with parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. The user must call **cusparseXbsrilu02_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsrilu02()** if **bsrilu02_analysis()** reports a structural zero. In this case, the user can still call **bsrilu02()**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.

bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.
info	structure initialized using **cusparseCreateBsrilu02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **bsrilu02_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**); the base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.11. cusparse<t>bsrilu02()

```
cusparseStatus_t
cusparseSbsrilu02(cusparseHandle_t handle,
                  cusparseDirection_t dirA,
                  int mb,
                  int nnzb,
                  const cusparseMatDescr_t descry,
                  float *bsrValA,
                  const int *bsrRowPtrA,
                  const int *bsrColIndA,
                  int blockDim,
                  bsrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseDbsrilu02(cusparseHandle_t handle,
                  cusparseDirection_t dirA,
                  int mb,
                  int nnzb,
                  const cusparseMatDescr_t descry,
                  double *bsrValA,
                  const int *bsrRowPtrA,
                  const int *bsrColIndA,
                  int blockDim,
                  bsrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseCbsrilu02(cusparseHandle_t handle,
                  cusparseDirection_t dirA,
                  int mb,
                  int nnzb,
                  const cusparseMatDescr_t descry,
                  cuComplex *bsrValA,
                  const int *bsrRowPtrA,
                  const int *bsrColIndA,
                  int blockDim,
                  bsrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);

cusparseStatus_t
cusparseZbsrilu02(cusparseHandle_t handle,
                  cusparseDirection_t dirA,
                  int mb,
                  int nnzb,
                  const cusparseMatDescr_t descry,
                  cuDoubleComplex *bsrValA,
                  const int *bsrRowPtrA,
                  const int *bsrColIndA,
                  int blockDim,
                  bsrilu02Info_t info,
                  cusparseSolvePolicy_t policy,
                  void *pBuffer);
```

This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrilu02()** supports an arbitrary **blockDim**.

This function requires a buffer size returned by **bsrilu02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrilu02()** can be used without level information, the user must be aware of consistency. If **bsrilu02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrilu02()** can be run with or without levels. On the other hand, if **bsrilu02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrilu02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrilu02()** has the same behavior as **csrilu02()**. That is, **bsr2csr(bsrilu02(A)) = csrilu02(bsr2csr(A))**. The numerical zero of **csrilu02()** means there exists some zero **U(j,j)**. The numerical zero of **bsrilu02()** means there exists some block **U(j,j)** that is not invertible.

Function **bsrilu02** reports the first numerical zero, including a structural zero. The user must call **cusparseXbsrilu02_zeroPivot()** to know where the numerical zero is.

For example, suppose **A** is a real m-by-m matrix where **m=mb*blockDim**. The following code solves precondition system **M*y = x**, where **M** is the product of LU factors **L** and **U**.

```
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
bsrilu02Info_t info_M = 0;
bsrsv2Info_t   info_L = 0;
bsrsv2Info_t   info_U = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);

cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsrilu02 and two info's for bsrsv2
cusparseCreateBsrilu02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_U);

// step 3: query how much memory used in bsrilu02 and bsrsv2, and allocate the
 buffer
cusparseDbsrilu02_bufferSize(handle, dir, mb, nnzb,
    descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M,
 &pBufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
 &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_U, mb, nnzb,
    descr_U, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_U,
```

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks: either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** + 1 elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb) - bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A; must be larger than zero.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **bsrilu02_bufferSize()**.

Output

bsrValA	<type> matrix containing the incomplete-LU lower and upper triangular factors.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**); the base index is not 0 or 1.
CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.2.12. cusparseXbsrilu02_zeroPivot()

```
cusparseStatus_t
cusparseXbsrilu02_zeroPivot(cusparseHandle_t handle,
                            bsrilu02Info_t info,
                            int *position);
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero (the block is not invertible). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsrilu02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set proper the mode with **cusparseSetPointerMode()**.

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **bsrilu02_analysis()** or **bsrilu02()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is not invertible, **position=j**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**info** is not valid.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.3. tridiagonal solve

Different algorithms for tridiagonal solve are discussed in this section.

10.3.1. cusparse<t>gtsv()

```
cusparseStatus_t
cusparseSgtsv(cusparseHandle_t handle,
              int m,
              int n,
              const float *dl,
              const float  *d,
              const float *du,
              float *B,
              int ldb)
cusparseStatus_t
cusparseDgtsv(cusparseHandle_t handle,
              int m,
              int n,
              const double *dl,
              const double  *d,
              const double *du,
              double *B,
              int ldb)
cusparseStatus_t
cusparseCgtsv(cusparseHandle_t handle,
              int m,
              int n,
              const cuComplex *dl,
              const cuComplex  *d,
              const cuComplex *du,
              cuComplex *B,
              int ldb)
cusparseStatus_t
cusparseZgtsv(cusparseHandle_t handle,
              int m,
              int n,
              const cuDoubleComplex *dl,
              const cuDoubleComplex  *d,
              const cuDoubleComplex *du,
              cuDoubleComplex *B,
              int ldb)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * Y = X$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-hand-side matrix **X** on exit.

Assuming **A** is of size **m** and base-1, **dl**, **d** and **du** are defined by the following formula:

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

The routine does perform pivoting, which usually results in more accurate and more stable results than **cusparse<t>gtsv_nopivot()** at the expense of some execution time

This routine requires significant amount of temporary extra storage (**min(m,8) ×(3+n)×sizeof(<type>)**). The temporary storage is managed by **cudaMalloc** and **cudaFree** which stop concurrency. The user can call **cusparse<t>gtsv2()** which has no explicit **cudaMalloc** and **cudaFree**.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B** (that is ≥ $\max(1, m)$).

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.3.2. cusparse<t>gtsv_nopivot()

```
cusparseStatus_t
cusparseSgtsv_nopivot(cusparseHandle_t handle, int m, int n,
              const float           *dl, const float           *d,
              const float           *du, float *B, int ldb)

cusparseStatus_t
cusparseDgtsv_nopivot(cusparseHandle_t handle, int m, int n,
              const double          *dl, const double          *d,
              const double          *du, double *B, int ldb)
cusparseStatus_t
cusparseCgtsv_nopivot(cusparseHandle_t handle, int m, int n,
              const cuComplex       *dl, const cuComplex       *d,
              const cuComplex       *du, cuComplex       *B, int ldb)
cusparseStatus_t
cusparseZgtsv_nopivot(cusparseHandle_t handle, int m, int n,
              const cuDoubleComplex *dl, const cuDoubleComplex  *d,
              const cuDoubleComplex *du, cuDoubleComplex *B, int ldb)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * Y = X$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-hand-side matrix **X** on exit.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This routine requires a significant amount of temporary extra storage (**m*(3+n)*sizeof(<type>)**). The temporary storage is managed by **cudaMalloc** and **cudaFree** which stop concurrency. The user can call **cusparse<t>gtsv2_nopivot()** which has no explicit **cudaMalloc** and **cudaFree**.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.

B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. (that is ≥ $\max(1, m)$).

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.3.3. cusparse<t>gtsv2_buffSizeExt()

```
cusparseStatus_t cusparseSgtsv2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *dl,
    const float  *d,
    const float *du,
    const float *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseDgtsv2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *dl,
    const double  *d,
    const double *du,
    const double *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseCgtsv2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    const cuComplex *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseZgtsv2_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    const cuDoubleComplex *B,
    int ldb,
    size_t *bufferSizeInBytes)
```

This function returns the size of the buffer used in **gtsv2** which computes the solution of a tridiagonal linear system with multiple right-hand sides.

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B** (that is ≥ $\max(1, m)$).

Output

pBufferSizeInBytes	number of bytes of the buffer used in the **gtsv2**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.3.4. cusparse<t>gtsv2()

```
cusparseStatus_t cusparseSgtsv2(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *dl,
    const float  *d,
    const float *du,
    float *B,
    int ldb,
    void *pBuffer)

cusparseStatus_t cusparseDgtsv2(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *dl,
    const double  *d,
    const double *du,
    double *B,
    int ldb,
    void *pBuffer)

cusparseStatus_t cusparseCgtsv2(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    cuComplex *B,
    int ldb,
    void *pBuffer)

cusparseStatus_t cusparseZgtsv2(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    cuDoubleComplex *B,
    int ldb,
    void *pBuffer)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Assuming **A** is of size **m** and base-1, **dl**, **d** and **du** are defined by the following formula:

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

The routine does perform pivoting, which usually results in more accurate and more stable results than **cusparse<t>gtsv_nopivot()** or **cusparse<t>gtsv2_nopivot()** at the expense of some execution time.

This function requires a buffer size returned by **gtsv2_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B** (that is ≥ $\max(1, m)$).
pBuffer	buffer allocated by the user, the size is return by **gtsv2_bufferSizeExt**.

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.

CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.3.5. cusparse<t>gtsv2_nopivot_bufferSizeExt()

```
cusparseStatus_t cusparseSgtsv2_nopivot_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *dl,
    const float  *d,
    const float *du,
    const float *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseDgtsv2_nopivot_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *dl,
    const double  *d,
    const double *du,
    const double *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseCgtsv2_nopivot_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    const cuComplex *B,
    int ldb,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseZgtsv2_nopivot_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    const cuDoubleComplex *B,
    int ldb,
    size_t *bufferSizeInBytes)
```

This function returns the size of the buffer used in **gtsv2_nopivot** which computes the solution of a tridiagonal linear system with multiple right-hand sides.

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. (that is ≥ $\max(1, m)$).

Output

pBufferSizeInBytes	number of bytes of the buffer used in the **gtsv2_nopivot**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.3.6. cusparse<t>gtsv2_nopivot()

```
cusparseStatus_t cusparseSgtsv2_nopivot(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *dl,
    const float  *d,
    const float *du,
    float *B,
    int ldb,
    void* pBuffer)

cusparseStatus_t cusparseDgtsv2_nopivot(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *dl,
    const double  *d,
    const double *du,
    double *B,
    int ldb,
    void* pBuffer)

cusparseStatus_t cusparseCgtsv2_nopivot(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    cuComplex *B,
    int ldb,
    void* pBuffer)

cusparseStatus_t cusparseZgtsv2_nopivot(
    cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    cuDoubleComplex *B,
    int ldb,
    void* pBuffer)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This function requires a buffer size returned by **gtsv2_nopivot_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. (that is ≥ $\max(1, m)$).
pBuffer	buffer allocated by the user, the size is return by **gtsv2_nopivot_bufferSizeExt**.

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.4. batched tridiagonal solve

Different algorithms for batched tridiagonal solve are discussed in this section.

10.4.1. cusparse<t>gtsvStridedBatch()

```
cusparseStatus_t
cusparseSgtsvStridedBatch(cusparseHandle_t handle, int m,
                          const float           *dl,
                          const float            *d,
                          const float           *du, float           *x,

                          int batchCount, int batchStride)
cusparseStatus_t
cusparseDgtsvStridedBatch(cusparseHandle_t handle, int m,
                          const double          *dl,
                          const double           *d,
                          const double          *du, double          *x,

                          int batchCount, int batchStride)

cusparseStatus_t
cusparseCgtsvStridedBatch(cusparseHandle_t handle, int m,
                          const cuComplex       *dl,
                          const cuComplex        *d,
                          const cuComplex       *du, cuComplex       *x,

                          int batchCount, int batchStride)
cusparseStatus_t
cusparseZgtsvStridedBatch(cusparseHandle_t handle, int m,
                          const cuDoubleComplex *dl,
                          const cuDoubleComplex  *d,
                          const cuDoubleComplex *du, cuDoubleComplex *x,

                          int batchCount, int batchStride)
```

This function computes the solution of multiple tridiagonal linear systems for *i*=0, …,**batchCount**:

$$A^{(i)} * y^{(i)} = x^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-hand-side matrix **X** on exit. The different matrices are assumed to be of the same size and are stored with a fixed **batchStride** in memory.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This routine requires a significant amount of temporary extra storage ((**batchCount**×**(4**×**m+2048)**×**sizeof(<type>)**)). The temporary storage is managed by **cudaMalloc** and **cudaFree** which stop concurrency. The user can call **cusparse<t>gtsv2StridedBatch()** which has no explicit **cudaMalloc** and **cudaFree**.

Input

handle	handle to the cuSPARSE library context.

m	the size of the linear system (must be ≥ 3).
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^{(i)}$ that corresponds to the i^{th} linear system starts at location **dl+batchStride×i** in memory. Also, the first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^{(i)}$ that corresponds to the i^{th} linear system starts at location **d+batchStride×i** in memory.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^{(i)}$ that corresponds to the i^{th} linear system starts at location **du+batchStride×i** in memory. Also, the last element of each upper diagonal must be zero.
x	<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.
batchCount	number of systems to solve.
batchStride	stride (number of elements) that separates the vectors of every system (must be at least **m**).

Output

x	<type> dense array that contains the solution of the tri-diagonal linear system. The solution $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, batchCount≤0, batchStride<m**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.4.2. cusparse<t>gtsv2StridedBatch_bufferSizeExt()

```
cusparseStatus_t cusparseSgtsv2StridedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    const float *dl,
    const float  *d,
    const float *du,
    const float *x,
    int batchCount,
    int batchStride,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseDgtsv2StridedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    const double *dl,
    const double  *d,
    const double *du,
    const double *x,
    int batchCount,
    int batchStride,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseCgtsv2StridedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    const cuComplex *x,
    int batchCount,
    int batchStride,
    size_t *bufferSizeInBytes)

cusparseStatus_t cusparseZgtsv2StridedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    const cuDoubleComplex *x,
    int batchCount,
    int batchStride,
    size_t *bufferSizeInBytes)
```

This function returns the size of the buffer used in **gtsv2StridedBatch** which computes the solution of multiple tridiagonal linear systems for i=0,…,**batchCount**:

$$A^{(i)} * y^{(i)} = x^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-

hand-side matrix **x** on exit. The different matrices are assumed to be of the same size and are stored with a fixed **batchStride** in memory.

Input

handle	handle to the cuSPARSE library context.
n	the size of the linear system (must be ≥ 3).
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^{(i)}$ that corresponds to the i^{th} linear system starts at location **dl+batchStride×i** in memory. Also, the first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^{(i)}$ that corresponds to the i^{th} linear system starts at location **d+batchStride×i** in memory.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^{(i)}$ that corresponds to the i^{th} linear system starts at location **du+batchStride×i** in memory. Also, the last element of each upper diagonal must be zero.
x	<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.
batchCount	number of systems to solve.
batchStride	stride (number of elements) that separates the vectors of every system (must be at least **m**).

Output

pBufferSizeInBytes	number of bytes of the buffer used in the **gtsv2StridedBatch**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, batchCount≤0, batchStride<m**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.4.3. cusparse<t>gtsv2StridedBatch()

```
cusparseStatus_t cusparseSgtsv2StridedBatch(
    cusparseHandle_t handle,
    int m,
    const float *dl,
    const float  *d,
    const float *du,
    float *x,
    int batchCount,
    int batchStride,
    void *pBuffer)

cusparseStatus_t cusparseDgtsv2StridedBatch(
    cusparseHandle_t handle,
    int m,
    const double *dl,
    const double  *d,
    const double *du,
    double *x,
    int batchCount,
    int batchStride,
    void *pBuffer)

cusparseStatus_t cusparseCgtsv2StridedBatch(
    cusparseHandle_t handle,
    int m,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    cuComplex *x,
    int batchCount,
    int batchStride,
    void *pBuffer)

cusparseStatus_t cusparseZgtsv2StridedBatch(
    cusparseHandle_t handle,
    int m,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    cuDoubleComplex *x,
    int batchCount,
    int batchStride,
    void *pBuffer)
```

This function computes the solution of multiple tridiagonal linear systems for *i*=0, …,**batchCount**:

$$A^{(i)} * y^{(i)} = x^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-

hand-side matrix **X** on exit. The different matrices are assumed to be of the same size and are stored with a fixed **batchStride** in memory.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This function requires a buffer size returned by **gtsv2StridedBatch_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Input

handle	handle to the cuSPARSE library context.
n	the size of the linear system (must be ≥ 3).
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^{(i)}$ that corresponds to the i^{th} linear system starts at location **dl+batchStride×i** in memory. Also, the first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^{(i)}$ that corresponds to the i^{th} linear system starts at location **d+batchStride×i** in memory.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^{(i)}$ that corresponds to the i^{th} linear system starts at location **du+batchStride×i** in memory. Also, the last element of each upper diagonal must be zero.
x	<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.
batchCount	number of systems to solve.
batchStride	stride (number of elements) that separates the vectors of every system (must be at least **n**).
pBuffer	buffer allocated by the user, the size is return by **gtsv2StridedBatch_bufferSizeExt**.

Output

x	<type> dense array that contains the solution of the tri-diagonal linear system. The solution $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.

CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<3, batchCount≤0, batchStride<m**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

10.4.4. cusparse<t>gtsvInterleavedBatch()

```
cusparseStatus_t
cusparseSgtsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const float *dl,
    const float  *d,
    const float *du,
    const float *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseDgtsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const double *dl,
    const double  *d,
    const double *du,
    const double *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseCgtsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    const cuComplex *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseZgtsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    const cuDoubleComplex *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseSgtsvInterleavedBatch(
    cusparseHandle_t handle,
    int algo,
    int m,
    float *dl,
    float  *d,
    float *du,
    float *x,
    int batchCount,
    void *pBuffer)

cusparseStatus_t
cusparseDgtsvInterleavedBatch(
    cusparseHandle_t handle,
```

This function computes the solution of multiple tridiagonal linear systems for *i*=0, …,**batchCount**:

$$A^{(i)} * x^{(i)} = b^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Assuming **A** is of size **m** and base-1, **dl**, **d** and **du** are defined by the following formula:

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

The data layout is different from **gtsvStridedBatch** which aggregates all matrices one after another. Instead, **gtsvInterleavedBatch** gathers different matrices of the same element in a continous manner. If **dl** is regarded as a 2-D array of size **m-by-batchCount**, **dl(:,j)** to store **j-th** matrix. **gtsvStridedBatch** uses column-major while **gtsvInterleavedBatch** uses row-major.

The routine provides three different algorithms, selected by parameter **algo**. The first algorithm is **cuThomas** provided by **Barcelona Supercomputing Center**. The second algorithm is LU with partial pivoting and last algorithm is QR. From stability perspective, cuThomas is not numerically stable because it does not have pivoting. LU with partial pivoting and QR are stable. From performance perspective, LU with partial pivoting and QR is about 10% to 20% slower than cuThomas.

This function requires a buffer size returned by **gtsvInterleavedBatch_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Appendix F shows an example of **gtsvInterleavedBatch**. If the user prepares aggregate format, one can use **cublasXgeam** to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

Input

handle	handle to the cuSPARSE library context.
algo	algo = 0: cuThomas (unstable algorithm); algo = 1: LU with pivoting (stable algorithm); algo = 2: QR (stable algorithm)
m	the size of the linear system.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.

d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
x	<type> dense right-hand-side array of dimensions **(batchCount, n)**.
pBuffer	buffer allocated by the user, the size is return by **gtsvInterleavedBatch_bufferSizeExt**.

Output

x	<type> dense solution array of dimensions **(batchCount, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**, **batchcount<0**).
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

10.5. batched pentadiagonal solve

Different algorithms for batched pentadiagonal solve are discussed in this section.

10.5.1. cusparse<t>gpsvInterleavedBatch()

```
cusparseStatus_t
cusparseSgpsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const float *ds,
    const float *dl,
    const float  *d,
    const float *du,
    const float *dw,
    const float *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseDgpsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const double *ds,
    const double *dl,
    const double  *d,
    const double *du,
    const double *dw,
    const double *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseCgpsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const cuComplex *ds,
    const cuComplex *dl,
    const cuComplex  *d,
    const cuComplex *du,
    const cuComplex *dw,
    const cuComplex *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseZgpsvInterleavedBatch_bufferSizeExt(
    cusparseHandle_t handle,
    int algo,
    int m,
    const cuDoubleComplex *ds,
    const cuDoubleComplex *dl,
    const cuDoubleComplex  *d,
    const cuDoubleComplex *du,
    const cuDoubleComplex *dw,
    const cuDoubleComplex *x,
    int batchCount,
    size_t *pBufferSizeInBytes)

cusparseStatus_t
cusparseSgpsvInterleavedBatch(
    cusparseHandle_t handle,
    int algo,
    int m,
    float *ds,
    float *dl,
```

This function computes the solution of multiple penta-diagonal linear systems for *i*=0, …,**batchCount**:

$$A^{(i)} * x^{(i)} = b^{(i)}$$

The coefficient matrix **A** of each of these penta-diagonal linear system is defined with five vectors corresponding to its lower (**ds, dl**), main (**d**), and upper (**du, dw**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Assuming **A** is of size **m** and base-1, **ds**, **dl**, **d**, **du** and **dw** are defined by the following formula:

ds(i) := A(i, i-2) for **i=1,2,...,m**

The first two elements of ds is out-of-bound (**ds(1) := A(1,-1)**, **ds(2) := A(2,0)**), so **ds(1) = 0** and **ds(2) = 0**.

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

dw(i) = A(i,i+2) for **i=1,2,...,m**

The last two elements of dw is out-of-bound (**dw(m-1) := A(m-1,m+1)**, **dw(m) := A(m,m+2)**), so **dw(m-1) = 0** and **dw(m) = 0**.

The data layout is the same as **gtsvStridedBatch**.

The routine is numerically stable because it uses QR to solve the linear system.

This function requires a buffer size returned by **gpsvInterleavedBatch_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Appendix G shows an example of **gpsvInterleavedBatch**. If the user prepares aggregate format, one can use **cublasXgeam** to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

Input

handle	handle to the cuSPARSE library context.
algo	only support algo = 0 (QR)
m	the size of the linear system.
ds	<type> dense array containing the lower diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The first two elements must be zero.

dl	<type> dense array containing the lower diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The first element must be zero.
d	<type> dense array containing the main diagonal of the penta-diagonal linear system.
du	<type> dense array containing the upper diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The last element must be zero.
dw	<type> dense array containing the upper diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The last two elements must be zero.
x	<type> dense right-hand-side array of dimensions **(batchCount, n)**.
pBuffer	buffer allocated by the user, the size is return by **gpsvInterleavedBatch_bufferSizeExt**.

Output

x	<type> dense solution array of dimensions **(batchCount, n)**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m<0**, **batchcount<0**).
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

Chapter 11. CUSPARSE REORDERINGS REFERENCE

This chapter describes the reordering routines used to manipulate sparse matrices.

11.1. cusparse<t>csrcolor()

```
cusparseStatus_t
cusparseScsrcolor(cusparseHandle_t handle, int m, int nnz,
             const cusparseMatDescr_t descrA, const float *csrValA,
             const int *csrRowPtrA, const int *csrColIndA,
             const float *fractionToColor, int *ncolors, int *coloring,
             int *reordering,cusparseColorInfo_t info);

cusparseStatus_t
cusparseDcsrcolor(cusparseHandle_t handle, int m, int nnz,
             const cusparseMatDescr_t descrA, const double *csrValA,
             const int *csrRowPtrA, const int *csrColIndA,
             const double *fractionToColor,int *ncolors, int *coloring,
             int *reordering, cusparseColorInfo_t info);

cusparseStatus_t
cusparseCcsrcolor(cusparseHandle_t handle, int m, int nnz,
             const cusparseMatDescr_t descrA, const cuComplex *csrValA,
             const int *csrRowPtrA, const int *csrColIndA,
             const float *fractionToColor, int *ncolors, int *coloring,
             int *reordering, cusparseColorInfo_t info);

cusparseStatus_t
cusparseZcsrcolor(cusparseHandle_t handle, int m, int nnz,
       const cusparseMatDescr_t descrA, const cuDoubleComplex *csrValA,
             const int *csrRowPtrA, const int *csrColIndA,
             const double *fractionToColor,int *ncolors, int *coloring,
             int *reordering, cusparseColorInfo_t info);
```

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format. The coloring is an assignment of colors (integer numbers) to nodes, such that neighboring nodes have distinct colors. An approximate coloring algorithm is used in this routine, and is stopped when a certain percentage of nodes has been colored. The rest of the nodes are assigned distinct colors (an increasing sequence of integers numbers, starting from the last integer used previously). The last two auxiliary routines can be used to extract the resulting number of colors, their assignment

and the associated reordering. The reordering is such that nodes that have been assigned the same color are reordered to be next to each other.

The matrix A passed to this routine, must be stored as a general matrix and have a symmetric sparsity pattern. If the matrix is nonsymmetric the user should pass A+A^T as a parameter to this routine.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
fractionToColor	fraction of nodes to be colored, which should be in the interval [0.0,1.0], for example 0.8 implies that 80 percent of nodes will be colored.
info	structure with information to be passed to the coloring.

Output

ncolors	The number of distinct colors used (at most the size of the matrix, but likely much smaller).
coloring	The resulting coloring permutation
reordering	The resulting reordering permutation (untouched if NULL)

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision (compute capability (c.c.) >= 1.3 required).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

Chapter 12. CUSPARSE FORMAT CONVERSION REFERENCE

This chapter describes the conversion routines between different sparse and dense storage formats.

coosort, **csrsort**, **cscsort**, **csru2csr** and **csr2csc_indexOnly** are sorting routines without malloc inside, the following table estimates the buffer size

routine	buffer size	maximum problem size if buffer is limited by 2GB
coosort	> 16*n bytes	125M
csrsort or cscsort	> 20*n bytes	100M
csru2csr	'd' > 28*n bytes ; 'z' > 36*n bytes	71M for 'd' and 55M for 'z'
csr2csc_indexOnl	> 16*n bytes	125M

12.1. cusparse<t>bsr2csr()

```
cusparseStatus_t
cusparseSbsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const float *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    const cusparseMatDescr_t descrC,
    float *csrValC,
    int *csrRowPtrC,
    int *csrColIndC)
cusparseStatus_t
cusparseDbsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const double *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    const cusparseMatDescr_t descrC,
    double *csrValC,
    int *csrRowPtrC,
    int *csrColIndC)
cusparseStatus_t
cusparseCbsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const cuComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    const cusparseMatDescr_t descrC,
    cuComplex *csrValC,
    int *csrRowPtrC,
    int *csrColIndC)
cusparseStatus_t
cusparseZbsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    const cusparseMatDescr_t descrC,
    cuDoubleComplex *csrValC,
    int *csrRowPtrC,
    int *csrColIndC)
```

This function converts a sparse matrix in BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**) into a sparse matrix in CSR format that is defined by arrays **csrValC**, **csrRowPtrC**, and **csrColIndC**.

Let **m(=mb*blockDim)** be the number of rows of **A** and **n(=nb*blockDim)** be number of columns of **A**, then **A** and **C** are **m*n** sparse matrices. The BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** nonzero blocks, whereas the sparse matrix **A** contains **nnz(=nnzb*blockDim*blockDim)** elements. The user must allocate enough space for arrays **csrRowPtrC**, **csrColIndC**, and **csrValC**. The requirements are as follows:

csrRowPtrC of **m+1** elements

csrValC of **nnz** elements

csrColIndC of **nnz** elements

The general procedure is as follows:

```
// Given BSR format (bsrRowPtrA, bsrcolIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb*blockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz  = nnzb * blockDim * blockDim; // number of elements
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSbsr2csr(handle, dir, mb, nb,
        descrA,
        bsrValA, bsrRowPtrA, bsrColIndA,
        blockDim,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**.
bsrValA	<type> array of **nnzb*blockDim*blockDim** nonzero elements of matrix **A**.
bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**.
descrC	the descriptor of matrix **C**.

Output

csrValC	<type> array of **nnz(=csrRowPtrC[m]-csrRowPtrC[0])** nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **C**.
csrColIndC	integer array of **nnz** column indices of the nonzero elements of matrix **C**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nb<0**, **IndexBase** of **descrA**, **descrC** is not base-0 or base-1, **dir** is not row-major or column-major, or **blockDim**<1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.2. cusparse<t>gebsr2gebsc_bufferSize()

```
cusparseStatus_t
cusparseSgebsr2gebsc_bufferSize(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const float *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseDgebsr2gebsc_bufferSize(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const double *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseCgebsr2gebsc_bufferSize(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const cuComplex *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseZgebsr2gebsc_bufferSize(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const cuDoubleComplex *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)
```

This function returns size of buffer used in computing **gebsr2gebsc()**.

Input

handle	handle to the cuSPARSE library context.
mb	number of block rows of sparse matrix **A**.

nb	number of block columns of sparse matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
bsrVal	<type> array of **nnzb*rowBlockDim*colBlockDim** non-zero elements of matrix **A**.
bsrRowPtr	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one.
bsrColInd	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.
rowBlockDim	number of rows within a block of **A**.
colBlockDim	number of columns within a block of **A**.

Output

pBufferSize	host pointer containing number of bytes of the buffer used in **gebsr2gebsc()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb, nb, nnzb<0**, or **rowBlockDim, colBlockDim**<1).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.3. cusparse<t>gebsr2gebsc()

```
cusparseStatus_t
cusparseSgebsr2gebsc(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const float *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    float *bscVal,
    int *bscRowInd,
    int *bscColPtr,
    cusparseAction_t copyValues,
    cusparseIndexBase_t baseIdx,
    void *pBuffer)

cusparseStatus_t
cusparseDgebsr2gebsc(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const double *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    double *bscVal,
    int *bscRowInd,
    int *bscColPtr,
    cusparseAction_t copyValues,
    cusparseIndexBase_t baseIdx,
    void *pBuffer)

cusparseStatus_t
cusparseCgebsr2gebsc(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const cuComplex *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    cuComplex *bscVal,
    int *bscRowInd,
    int *bscColPtr,
    cusparseAction_t copyValues,
    cusparseIndexBase_t baseIdx,
    void *pBuffer)

cusparseStatus_t
cusparseZgebsr2gebsc(cusparseHandle_t handle,
    int mb,
    int nb,
    int nnzb,
    const cuDoubleComplex *bsrVal,
    const int *bsrRowPtr,
    const int *bsrColInd,
    int rowBlockDim,
    int colBlockDim,
    cuDoubleComplex *bscVal,
    int *bscRowInd,
```

This function can be seen as the same as **csr2csc()** when each block of size **rowBlockDim*colBlockDim** is regarded as a scalar.

This sparsity pattern of the result matrix can also be seen as the transpose of the original sparse matrix, but the memory layout of a block does not change.

The user must call **gebsr2gebsc_bufferSize()** to determine the size of the buffer required by **gebsr2gebsc()**, allocate the buffer, and pass the buffer pointer to **gebsr2gebsc()**.

Input

handle	handle to the cuSPARSE library context.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
bsrVal	<type> array of **nnzb*rowBlockDim*colBlockDim** nonzero elements of matrix **A**.
bsrRowPtr	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one.
bsrColInd	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.
rowBlockDim	number of rows within a block of **A**.
colBlockDim	number of columns within a block of **A**.
copyValues	**CUSPARSE_ACTION_SYMBOLIC** or **CUSPARSE_ACTION_NUMERIC**.
baseIdx	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	buffer allocated by the user; the size is return by **gebsr2gebsc_bufferSize()**.

Output

bscVal	<type> array of **nnzb*rowBlockDim*colBlockDim** non-zero elements of matrix **A**. It is only filled-in if **copyValues** is set to **CUSPARSE_ACTION_NUMERIC**.
bscRowInd	integer array of **nnzb** row indices of the non-zero blocks of matrix **A**.
bscColPtr	integer array of **nb+1** elements that contains the start of every block column and the end of the last block column plus one.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.

CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nb,nnzb<0**, **baseIdx** is not base-0 or base-1, or **rowBlockDim**, **colBlockDim**<1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.4. cusparse<t>gebsr2gebsr_bufferSize()

```
cusparseStatus_t
cusparseSgebsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const float *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    int rowBlockDimC,
    int colBlockDimC,
    int *pBufferSize )

cusparseStatus_t
cusparseDgebsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const double *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    int rowBlockDimC,
    int colBlockDimC,
    int *pBufferSize )

cusparseStatus_t
cusparseCgebsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const cuComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    int rowBlockDimC,
    int colBlockDimC,
    int *pBufferSize )

cusparseStatus_t
cusparseZgebsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    int rowBlockDimC,
    int colBlockDimC,
    int *pBufferSize )
```

This function returns size of the buffer used in computing **gebsr2gebsrNnz()** and **gebsr2gebsr()**.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb*rowBlockDimA*colBlockDimA** non-zero elements of matrix **A**.
bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the nonzero blocks of matrix **A**.
rowBlockDimA	number of rows within a block of **A**.
colBlockDimA	number of columns within a block of **A**.
rowBlockDimC	number of rows within a block of C.
colBlockDimC	number of columns within a block of C

Output

pBufferSize	host pointer containing number of bytes of the buffer used in **gebsr2gebsr()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb**, **nb**, **nnzb**<0; or **rowBlockDimA**, **colBlockDimA**, **rowBlockDimC**, **colBlockDimC**<1).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.5. cusparse<t>gebsr2gebsr()

```
cusparseStatus_t
cusparseXgebsr2gebsrNnz(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    const cusparseMatDescr_t descrC,
    int *bsrRowPtrC,
    int rowBlockDimC,
    int colBlockDimC,
    int *nnzTotalDevHostPtr,
    void *pBuffer)

cusparseStatus_t
cusparseSgebsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const float *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    const cusparseMatDescr_t descrC,
    float *bsrValC,
    int *bsrRowPtrC,
    int *bsrColIndC,
    int rowBlockDimC,
    int colBlockDimC,
    void *pBuffer)

cusparseStatus_t
cusparseDgebsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    const double *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int rowBlockDimA,
    int colBlockDimA,
    const cusparseMatDescr_t descrC,
    double *bsrValC,
    int *bsrRowPtrC,
    int *bsrColIndC,
    int rowBlockDimC,
    int colBlockDimC,
    void *pBuffer)

cusparseStatus_t
cusparseCgebsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    int nnzb,
```

This function converts a sparse matrix in general BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA** into a sparse matrix in another general BSR format that is defined by arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**.

If **rowBlockDimA=1** and **colBlockDimA=1**, **cusparse[S|D|C|Z]gebsr2gebsr()** is the same as **cusparse[S|D|C|Z]csr2gebsr()**.

If **rowBlockDimC=1** and **colBlockDimC=1**, **cusparse[S|D|C|Z]gebsr2gebsr()** is the same as **cusparse[S|D|C|Z]gebsr2csr()**.

A is an **m*n** sparse matrix where **m(=mb*rowBlockDim)** is the number of rows of **A**, and **n(=nb*colBlockDim)** is the number of columns of **A**. The general BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** nonzero blocks. The matrix **C** is also general BSR format with a different block size, **rowBlockDimC*colBlockDimC**. If **m** is not a multiple of **rowBlockDimC**, or **n** is not a multiple of **colBlockDimC**, zeros are filled in. The number of block rows of **C** is **mc(=(m+rowBlockDimC-1)/rowBlockDimC)**. The number of block rows of **C** is **nc(=(n+colBlockDimC-1)/colBlockDimC)**. The number of nonzero blocks of **C** is **nnzc**.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **bsrRowPtrC** of **mc+1** elements and uses function **cusparseXgebsr2gebsrNnz()** to determine the number of nonzero block columns per block row of matrix **C**. Second, the user gathers **nnzc** (number of non-zero block columns of matrix **C**) from either **(nnzc=*nnzTotalDevHostPtr)** or **(nnzc=bsrRowPtrC[mc]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzc*rowBlockDimC*colBlockDimC** elements and **bsrColIndC** of **nnzc** integers. Finally the function **cusparse[S|D|C|Z]gebsr2gebsr()** is called to complete the conversion.

The user must call **gebsr2gebsr_bufferSize()** to know the size of the buffer required by **gebsr2gebsr()**, allocate the buffer, and pass the buffer pointer to **gebsr2gebsr()**.

The general procedure is as follows:

```
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzc;
int m = mb*rowBlockDimA;
int n = nb*colBlockDimA;
int mc = (m+rowBlockDimC-1)/rowBlockDimC;
int nc = (n+colBlockDimC-1)/colBlockDimC;
int bufferSize;
void *pBuffer;
cusparseSgebsr2gebsr_bufferSize(handle, dir, mb, nb, nnzb,
    descrA, bsrValA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    rowBlockDimC, colBlockDimC,
    &bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int)*(mc+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzc;
cusparseXgebsr2gebsrNnz(handle, dir, mb, nb, nnzb,
    descrA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    descrC, bsrRowPtrC,
    rowBlockDimC, colBlockDimC,
    nnzTotalDevHostPtr,
    pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzc = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzc, bsrRowPtrC+mc, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzc -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzc);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDimC*colBlockDimC)*nnzc);
cusparseSgebsr2gebsr(handle, dir, mb, nb, nnzb,
    descrA, bsrValA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    descrC, bsrValC, bsrRowPtrC, bsrColIndC,
    rowBlockDimC, colBlockDimC,
    pBuffer);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb*rowBlockDimA*colBlockDimA** non-zero elements of matrix **A**.

bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.
rowBlockDimA	number of rows within a block of **A**.
colBlockDimA	number of columns within a block of **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
rowBlockDimC	number of rows within a block of **C**.
colBlockDimC	number of columns within a block of **C**.
pBuffer	buffer allocated by the user; the size is return by **gebsr2gebsr_bufferSize()**.

Output

bsrValC	<type> array of **nnzc*rowBlockDimC*colBlockDimC** non-zero elements of matrix **C**.
bsrRowPtrC	integer array of **mc+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **C**.
bsrColIndC	integer array of **nnzc** block column indices of the nonzero blocks of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero blocks of **C**. ***nnzTotalDevHostPtr** is the same as **bsrRowPtrC[mc]-bsrRowPtrC[0]**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb, nb, nnzb**<0, **baseIdx** is not base-0 or base-1; or **rowBlockDimA**, **colBlockDimA**, **rowBlockDimC**, **colBlockDimC**<1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.6. cusparse<t>gebsr2csr()

```
cusparseStatus_t
cusparseSgebsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const float *bsrValA,
    const int    *bsrRowPtrA,
    const int    *bsrColIndA,
    int   rowBlockDim,
    int   colBlockDim,
    const cusparseMatDescr_t descrC,
    float  *csrValC,
    int    *csrRowPtrC,
    int    *csrColIndC )
cusparseStatus_t
cusparseDgebsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const double *bsrValA,
    const int    *bsrRowPtrA,
    const int    *bsrColIndA,
    int   rowBlockDim,
    int   colBlockDim,
    const cusparseMatDescr_t descrC,
    double  *csrValC,
    int    *csrRowPtrC,
    int    *csrColIndC )
cusparseStatus_t
cusparseCgebsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const cuComplex *bsrValA,
    const int    *bsrRowPtrA,
    const int    *bsrColIndA,
    int   rowBlockDim,
    int   colBlockDim,
    const cusparseMatDescr_t descrC,
    cuComplex  *csrValC,
    int    *csrRowPtrC,
    int    *csrColIndC )
cusparseStatus_t
cusparseZgebsr2csr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int mb,
    int nb,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *bsrValA,
    const int    *bsrRowPtrA,
    const int    *bsrColIndA,
    int   rowBlockDim,
    int   colBlockDim,
    const cusparseMatDescr_t descrC,
    cuDoubleComplex  *csrValC,
    int    *csrRowPtrC,
    int    *csrColIndC )
```

This function converts a sparse matrix in general BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA** into a sparse matrix in CSR format that is defined by arrays **csrValC**, **csrRowPtrC**, and **csrColIndC**.

Let **m(=mb*rowBlockDim)** be number of rows of **A** and **n(=nb*colBlockDim)** be number of columns of **A**, then **A** and **C** are **m*n** sparse matrices. The general BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** non-zero blocks, whereas sparse matrix **A** contains **nnz(=nnzb*rowBlockDim*colBlockDim)** elements. The user must allocate enough space for arrays **csrRowPtrC**, **csrColIndC**, and **csrValC**. The requirements are as follows:

csrRowPtrC of **m+1** elements

csrValC of **nnz** elements

csrColIndC of **nnz** elements

The general procedure is as follows:

```
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb*rowBlockDim;
int n = nb*colBlockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz  = nnzb * rowBlockDim * colBlockDim; // number of elements
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSgebsr2csr(handle, dir, mb, nb,
        descrA,
        bsrValA, bsrRowPtrA, bsrColIndA,
        rowBlockDim, colBlockDim,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb*rowBlockDim*colBlockDim** non-zero elements of matrix **A**.
bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.

rowBlockDim	number of rows within a block of **A**.
colBlockDim	number of columns within a block of **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.

Output

csrValC	<type> array of **nnz** non-zero elements of matrix **C**.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **C**.
csrColIndC	integer array of **nnz** column indices of the non-zero elements of matrix **C**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb, nb<0** is not base-0 or base-1, or **rowBlockDim**, **colBlockDim**<1).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.7. cusparse<t>csr2gebsr_bufferSize()

```
cusparseStatus_t
cusparseScsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseDcsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseCcsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)

cusparseStatus_t
cusparseZcsr2gebsr_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    int rowBlockDim,
    int colBlockDim,
    int *pBufferSize)
```

This function returns the size of the buffer used in computing **csr2gebsrNnz** and **csr2gebsr**.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of sparse matrix **A**.
n	number of columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **A**.
csrColIndA	integer array of **nnz** column indices of the nonzero elements of matrix **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
rowBlockDim	number of rows within a block of **C**.
colBlockDim	number of columns within a block of **C**.

Output

pBufferSize	host pointer containing number of bytes of the buffer used in **csr2gebsrNnz()** and **csr2gebsr()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n<0**, or **rowBlockDim**, **colBlockDim<1**).
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.8. cusparse<t>csr2gebsr()

```
cusparseStatus_t
cusparseXcsr2gebsrNnz(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cusparseMatDescr_t descrC,
    int *bsrRowPtrC,
    int rowBlockDim,
    int colBlockDim,
    int *nnzTotalDevHostPtr,
    void *pBuffer )

cusparseStatus_t
cusparseScsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cusparseMatDescr_t descrC,
    float *bsrValC,
    int *bsrRowPtrC,
    int *bsrColIndC,
    int rowBlockDim,
    int colBlockDim,
    void *pBuffer)

cusparseStatus_t
cusparseDcsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cusparseMatDescr_t descrC,
    double *bsrValC,
    int *bsrRowPtrC,
    int *bsrColIndC,
    int rowBlockDim,
    int colBlockDim,
    void *pBuffer)

cusparseStatus_t
cusparseCcsr2gebsr(cusparseHandle_t handle,
    cusparseDirection_t dir,
    int m,
    int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const cusparseMatDescr_t descrC,
    cuComplex *bsrValC,
    int *bsrRowPtrC,
    int *bsrColIndC,
    int rowBlockDim,
```

This function converts a sparse matrix **A** in CSR format (that is defined by arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**) into a sparse matrix **C** in general BSR format (that is defined by the three arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**).

The matrix A is a **m*n** sparse matrix and matrix **C** is a **(mb*rowBlockDim)*(nb*colBlockDim)** sparse matrix, where **mb(=(m +rowBlockDim-1)/rowBlockDim)** is the number of block rows of **C**, and **nb(=(n +colBlockDim-1)/colBlockDim)** is the number of block columns of **C**.

The block of **C** is of size **rowBlockDim*colBlockDim**. If **m** is not multiple of **rowBlockDim** or **n** is not multiple of **colBlockDim**, zeros are filled in.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **bsrRowPtrC** of **mb+1** elements and uses function **cusparseXcsr2gebsrNnz()** to determine the number of nonzero block columns per block row. Second, the user gathers **nnzb** (number of nonzero block columns of matrix **C**) from either **(nnzb=*nnzTotalDevHostPtr)** or **(nnzb=bsrRowPtrC[mb]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzb*rowBlockDim*colBlockDim** elements and **bsrColIndC** of **nnzb** integers. Finally function **cusparse[S|D|C|Z]csr2gebsr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **csr2gebsr()** by calling **csr2gebsr_bufferSize()**, allocate the buffer, and pass the buffer pointer to **csr2gebsr()**.

The general procedure is as follows:

```
// Given CSR format (csrRowPtrA, csrColIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + rowBlockDim-1)/rowBlockDim;
int nb = (n + colBlockDim-1)/colBlockDim;
int bufferSize;
void *pBuffer;
cusparseScsr2gebsr_bufferSize(handle, dir, m, n,
    descrA, csrValA, csrRowPtrA, csrColIndA,
    rowBlockDim, colBlockDim,
    &bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2gebsrNnz(handle, dir, m, n,
    descrA, csrRowPtrA, csrColIndA,
    descrC, bsrRowPtrC, rowBlockDim, colBlockDim,
    nnzTotalDevHostPtr,
    pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzb = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzb, bsrRowPtrC+mb, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDim*colBlockDim)*nnzb);
cusparseScsr2gebsr(handle, dir, m, n,
        descrA,
        csrValA, csrRowPtrA, csrColIndA,
        descrC,
        bsrValC, bsrRowPtrC, bsrColIndC,
        rowBlockDim, colBlockDim,
        pBuffer);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of sparse matrix **A**.
n	number of columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **A**.
csrColIndA	integer array of **nnz** column indices of the nonzero elements of matrix **A**.

descrC	the descriptor of matrix C. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
rowBlockDim	number of rows within a block of C.
colBlockDim	number of columns within a block of C.
pBuffer	buffer allocated by the user, the size is return by **csr2gebsr_bufferSize()**.

Output

bsrValC	<type> array of **nnzb*rowBlockDim*colBlockDim** nonzero elements of matrix C.
bsrRowPtrC	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix C.
bsrColIndC	integer array of **nnzb** column indices of the nonzero blocks of matrix C.
nnzTotalDevHostPtr	total number of nonzero blocks of matrix C. Pointer **nnzTotalDevHostPtr** can point to a device memory or host memory.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n<0**, **baseIdx** is not base-0 or base-1, or **rowBlockDim**, **colBlockDim<1**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.9. cusparse<t>coo2csr()

```
cusparseStatus_t
cusparseXcoo2csr(cusparseHandle_t handle, const int *cooRowInd,
                 int nnz, int m, int *csrRowPtr, cusparseIndexBase_t
 idxBase)
```

This function converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).

It can also be used to convert the array containing the uncompressed column indices (corresponding to COO format) into an array of column pointers (corresponding to CSC format).

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
cooRowInd	integer array of **nnz** uncompressed row indices.
nnz	number of non-zeros of the sparse matrix (that is also the length of array **cooRowInd**).
m	number of rows of matrix **A**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

csrRowPtr	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

12.10. cusparse<t>csc2dense()

```
cusparseStatus_t
cusparseScsc2dense(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const float           *cscValA,
                   const int *cscRowIndA, const int *cscColPtrA,
                   float           *A, int lda)
cusparseStatus_t
cusparseDcsc2dense(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const double          *cscValA,
                   const int *cscRowIndA, const int *cscColPtrA,
                  double          *A, int lda)
cusparseStatus_t
cusparseCcsc2dense(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex       *cscValA,
                   const int *cscRowIndA, const int *cscColPtrA,
                   cuComplex       *A, int lda)
cusparseStatus_t
cusparseZcsc2dense(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex *cscValA,
                   const int *cscRowIndA, const int *cscColPtrA,
                   cuDoubleComplex *A, int lda)
```

This function converts the sparse matrix in CSC format that is defined by the three arrays **cscValA**, **cscColPtrA**, and **cscRowIndA** into the matrix **A** in dense format. The dense matrix **A** is filled in with the values of the sparse matrix and with zeros elsewhere.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
cscValA	<type> array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) nonzero elements of matrix **A**.
cscRowIndA	integer array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) row indices of the nonzero elements of matrix **A**.
cscColPtrA	integer array of **n+1** elements that contains the start of every row and the end of the last column plus one.
lda	leading dimension of dense array **A**.

Output

A	array of dimensions **(lda, n)** that is filled in with the values of the sparse matrix.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.11. cusparse<t>csc2hyb()

```
cusparseStatus_t
cusparseScsc2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const float           *cscValA,
            const int *cscRowIndA, const int *cscColPtrA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseDcsc2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const double          *cscValA,
            const int *cscRowIndA, const int *cscColPtrA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseCcsc2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const cuComplex       *cscValA,
            const int *cscRowIndA, const int *cscColPtrA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseZcsc2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const cuDoubleComplex *cscValA,
            const int *cscRowIndA, const int *cscColPtrA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
```

This function converts a sparse matrix in CSC format into a sparse matrix in HYB format. It assumes that the **hybA** parameter has been initialized with the **cusparseCreateHybMat()** routine before calling this function.

This function requires some amount of temporary storage and a significant amount of storage for the matrix in HYB format. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
cscValA	<type> array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) nonzero elements of matrix **A**.
cscRowIndA	integer array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) column indices of the nonzero elements of matrix **A**.
cscColPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
userEllWidth	width of the regular (ELL) part of the matrix in HYB format, which should be less than the maximum number of nonzeros per row and is only required if **partitionType** == **CUSPARSE_HYB_PARTITION_USER**.
partitionType	partitioning method to be used in the conversion (please refer to **cusparseHybPartition_t** for details).

Output

hybA	the matrix **A** in HYB storage format.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.12. cusparse<t>csr2bsr()

```
cusparseStatus_t
cusparseXcsr2bsrNnz(cusparseHandle_t handle,
      cusparseDirection_t dir,
      int m,
      int n,
      const cusparseMatDescr_t descrA,
      const int *csrRowPtrA,
      const int *csrColIndA,
      int blockDim,
      const cusparseMatDescr_t descrC,
      int *bsrRowPtrC,
      int *nnzTotalDevHostPtr)
cusparseStatus_t
cusparseScsr2bsr(cusparseHandle_t handle,
      cusparseDirection_t dir,
      int m,
      int n,
      const cusparseMatDescr_t descrA,
      const float *csrValA,
      const int *csrRowPtrA,
      const int *csrColIndA,
      int blockDim,
      const cusparseMatDescr_t descrC,
      float *bsrValC,
      int *bsrRowPtrC,
      int *bsrColIndC)
cusparseStatus_t
cusparseDcsr2bsr(cusparseHandle_t handle,
      cusparseDirection_t dir,
      int m,
      int n,
      const cusparseMatDescr_t descrA,
      const double *csrValA,
      const int *csrRowPtrA,
      const int *csrColIndA,
      int blockDim,
      const cusparseMatDescr_t descrC,
      double *bsrValC,
      int *bsrRowPtrC,
      int *bsrColIndC)
cusparseStatus_t
cusparseCcsr2bsr(cusparseHandle_t handle,
      cusparseDirection_t dir,
      int m,
      int n,
      const cusparseMatDescr_t descrA,
      const cuComplex *csrValA,
      const int *csrRowPtrA,
      const int *csrColIndA,
      int blockDim,
      const cusparseMatDescr_t descrC,
      cuComplex *bsrValC,
      int *bsrRowPtrC,
      int *bsrColIndC)
cusparseStatus_t
cusparseZcsr2bsr(cusparseHandle_t handle,
      cusparseDirection_t dir,
      int m,
      int n,
      const cusparseMatDescr_t descrA,
      const cuDoubleComplex *csrValA,
      const int *csrRowPtrA,
      const int *csrColIndA,
      int blockDim,
```

This function converts a sparse matrix in CSR format that is defined by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA** into a sparse matrix in BSR format that is defined by arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**.

A is an **m*n** sparse matrix. The BSR format of **A** has **mb** block rows, **nb** block columns, and **nnzb** nonzero blocks, where **mb=((m+blockDim-1)/blockDim)** and **nb=(n+blockDim-1)/blockDim**.

If **m** or **n** is not multiple of **blockDim**, zeros are filled in.

The conversion in cuSPARSE entails a two-step approach. First, the user allocates **bsrRowPtrC** of **mb+1** elements and uses function **cusparseXcsr2bsrNnz()** to determine the number of nonzero block columns per block row. Second, the user gathers **nnzb** (number of non-zero block columns of matrix **C**) from either **(nnzb=*nnzTotalDevHostPtr)** or **(nnzb=bsrRowPtrC[mb]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzb*blockDim*blockDim** elements and **bsrColIndC** of **nnzb** elements. Finally function **cusparse[S|D|C|Z]csr2bsr90** is called to complete the conversion.

The general procedure is as follows:

```
// Given CSR format (csrRowPtrA, csrcolIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2bsrNnz(handle, dir, m, n,
        descrA, csrRowPtrA, csrColIndA,
        blockDim,
        descrC, bsrRowPtrC,
        nnzTotalDevHostPtr);
if (NULL != nnzTotalDevHostPtr){
    nnzb = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzb, bsrRowPtrC+mb, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dir, m, n,
        descrA,
        csrValA, csrRowPtrA, csrColIndA,
        blockDim,
        descrC,
        bsrValC, bsrRowPtrC, bsrColIndC);
```

If **blockDim** is large (typically, a block cannot fit into shared memory), **cusparse[S|D|C|Z]csr2bsr()** allocates a temporary integer array of size **mb*blockDim** integers. If device memory is not available, **CUSPARSE_STATUS_ALLOC_FAILED** is returned.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.

m	number of rows of sparse matrix **A**.
n	number of columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**.
csrValA	<type> array of **nnz (=csrRowPtrA[m] - csrRowPtr[0])** non-zero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** column indices of the non-zero elements of matrix **A**.
blockDim	block dimension of sparse matrix **A**. The range of **blockDim** is between 1 and **min(m,n)**.
descrC	the descriptor of matrix **C**.

Output

bsrValC	<type> array of **nnzb*blockDim*blockDim** nonzero elements of matrix **C**.
bsrRowPtrC	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **C**.
bsrColIndC	integer array of **nnzb** column indices of the non-zero blocks of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(bsrRowPtrC[mb] - bsrRowPtrC[0])**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**). **IndexBase** field of **descrA, descrC** is not base-0 or base-1, **dir** is not row-major or column-major, or **blockDim** is not between 1 and **min(m,n)**.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.13. cusparse<t>csr2coo()

```
cusparseStatus_t
cusparseXcsr2coo(cusparseHandle_t handle, const int *csrRowPtr,
                 int nnz, int m, int *cooRowInd,
                 cusparseIndexBase_t idxBase)
```

This function converts the array containing the compressed row pointers (corresponding to CSR format) into an array of uncompressed row indices (corresponding to COO format).

It can also be used to convert the array containing the compressed column indices (corresponding to CSC format) into an array of uncompressed column indices (corresponding to COO format).

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
csrRowPtr	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
nnz	number of nonzeros of the sparse matrix (that is also the length of array **cooRowInd**).
m	number of rows of matrix **A**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

cooRowInd	integer array of **nnz** uncompressed row indices.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	**idxBase** is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.

12.14. cusparse<t>csr2csc()

```
cusparseStatus_t
cusparseScsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
                 const float *csrVal, const int *csrRowPtr,
                 const int *csrColInd, float           *cscVal,
                 int *cscRowInd, int *cscColPtr,
                 cusparseAction_t copyValues,
                 cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseDcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
                 const double *csrVal, const int *csrRowPtr,
                 const int *csrColInd, double          *cscVal,
                 int *cscRowInd, int *cscColPtr,
                 cusparseAction_t copyValues,
                 cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseCcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
                 const cuComplex *csrVal, const int *csrRowPtr,
                 const int *csrColInd, cuComplex       *cscVal,
                 int *cscRowInd, int *cscColPtr,
                 cusparseAction_t copyValues,
                 cusparseIndexBase_t idxBase)
cusparseStatus_t
cusparseZcsr2csc(cusparseHandle_t handle, int m, int n, int nnz,
                 const cuDoubleComplex *csrVal, const int *csrRowPtr,
                 const int *csrColInd, cuDoubleComplex *cscVal,
                 int *cscRowInd, int *cscColPtr,
                 cusparseAction_t copyValues,
                 cusparseIndexBase_t idxBase)
```

This function converts a sparse matrix in CSR format (that is defined by the three arrays **csrVal**, **csrRowPtr**, and **csrColInd**) into a sparse matrix in CSC format (that is defined by arrays **cscVal**, **cscRowInd**, and **cscColPtr**). The resulting matrix can also be seen as the transpose of the original sparse matrix. Notice that this routine can also be used to convert a matrix in CSC format into a matrix in CSR format.

This function requires a significant amount of extra storage that is proportional to the matrix size. It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix A.
n	number of columns of matrix A.
nnz	number of nonzero elements of matrix A.
csrVal	<type> array of **nnz** (= **csrRowPtr(m)** - **csrRowPtr(0)**) nonzero elements of matrix A.
csrRowPtr	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.

csrColInd	integer array of **nnz** (= **csrRowPtr(m) - csrRowPtr(0)**) column indices of the nonzero elements of matrix A.
copyValues	**CUSPARSE_ACTION_SYMBOLIC** or **CUSPARSE_ACTION_NUMERIC**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

cscVal	<type> array of **nnz** (= **cscColPtr(n) - cscColPtr(0)**) nonzero elements of matrix A. It is only filled in if **copyValues** is set to **CUSPARSE_ACTION_NUMERIC**.
cscRowInd	integer array of **nnz** (= **cscColPtr(n) - cscColPtr(0)**) column indices of the nonzero elements of matrix A.
cscColPtr	integer array of **n+1** elements that contains the start of every column and the end of the last column plus one.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.15. cusparseCsr2cscEx()

```
cusparseStatus_t cusparseCsr2cscEx(cusparseHandle_t handle,
                                   int m, int n, int nnz,
                                   const void  *csrSortedVal,
 cudaDataType csrSortedValtype,
                                   const int *csrSortedRowPtr,
                                   const int *csrSortedColInd,
                                   void *cscSortedVal, cudaDataType
 cscSortedValtype,
                                   int *cscSortedRowInd,
                                   int *cscSortedColPtr,
                                   cusparseAction_t copyValues,
                                   cusparseIndexBase_t idxBase,
                                   cudaDataType executiontype);
```

This function is an extended version of **cusparse<t>csr2csc()**. For detailed description of the functionality, see **cusparse<t>csr2csc()**.

This function does not support half-precision execution type, but it supports half-precision IO with single precision execution.

Input specifically required by `cusparseCsr2cscEx`

`csrSortedValAtype`	Data type of **`csrSortedValA`**.
`cscSortedValAtype`	Data type of **`cscSortedValA`**.
`executiontype`	Data type used for computation.

12.16. cusparse<t>csr2dense()

```
cusparseStatus_t cusparseScsr2dense(cusparseHandle_t handle,
                                    int m,
                                    int n,
                                    const cusparseMatDescr_t descrA,
                                    const float *csrValA,
                                    const int *csrRowPtrA,
                                    const int *csrColIndA,
                                    float *A,
                                    int lda)

cusparseStatus_t cusparseDcsr2dense(cusparseHandle_t handle,
                                    int m,
                                    int n,
                                    const cusparseMatDescr_t descrA,
                                    const double *csrValA,
                                    const int *csrRowPtrA,
                                    const int *csrColIndA,
                                    double *A,
                                    int lda)

cusparseStatus_t cusparseCcsr2dense(cusparseHandle_t handle,
                                    int m,
                                    int n,
                                    const cusparseMatDescr_t descrA,
                                    const cuComplex *csrValA,
                                    const int *csrRowPtrA,
                                    const int *csrColIndA,
                                    cuComplex *A,
                                    int lda)

cusparseStatus_t cusparseZcsr2dense(cusparseHandle_t handle,
                                    int m,
                                    int n,
                                    const cusparseMatDescr_t descrA,
                                    const cuDoubleComplex *csrValA,
                                    const int *csrRowPtrA,
                                    const int *csrColIndA,
                                    cuDoubleComplex *A,
                                    int lda)
```

This function converts the sparse matrix in CSR format (that is defined by the three arrays **`csrValA`**, **`csrRowPtrA`**, and **`csrColIndA`**) into the matrix **`A`** in dense format. The dense matrix **`A`** is filled in with the values of the sparse matrix and with zeros elsewhere.

This function requires no extra storage. It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix A.
n	number of columns of matrix A.
descrA	the descriptor of matrix A. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix A.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix A.
lda	leading dimension of array matrix**A**.

Output

A	array of dimensions **(lda,n)** that is filled in with the values of the sparse matrix.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.17. cusparse<t>csr2csr_compress()

```
cusparseStatus_t cusparseScsr2csr_compress(cusparseHandle_t handle,
                                           int m,
                                           int n,
                                           const cusparseMatDescr_t descrA,
                                           const float *csrValA,
                                           const int *csrColIndA,
                                           const int *csrRowPtrA,
                                           int nnzA,
                                           const int *nnzPerRow,
                                           float *csrValC,
                                           int *csrColIndC,
                                           int *csrRowPtrC,
                                           float tol);

cusparseStatus_t cusparseDcsr2csr_compress(cusparseHandle_t handle,
                                           int m,
                                           int n,
                                           const cusparseMatDescr_t descrA,
                                           const double *csrValA,
                                           const int *csrColIndA,
                                           const int *csrRowPtrA,
                                           int nnzA,
                                           const int *nnzPerRow,
                                           double *csrValC,
                                           int *csrColIndC,
                                           int *csrRowPtrC,
                                           double tol);

cusparseStatus_t cusparseCcsr2csr_compress(cusparseHandle_t handle,
                                           int m,
                                           int n,
                                           const cusparseMatDescr_t descrA,
                                           const cuComplex *csrValA,
                                           const int *csrColIndA,
                                           const int *csrRowPtrA,
                                           int nnzA,
                                           const int *nnzPerRow,
                                           cuComplex *csrValC,
                                           int *csrColIndC,
                                           int *csrRowPtrC,
                                           cuComplex tol);

cusparseStatus_t cusparseZcsr2csr_compress(cusparseHandle_t handle,
                                           int m,
                                           int n,
                                           const cusparseMatDescr_t descrA,
                                           const cuDoubleComplex *csrValA,
                                           const int *csrColIndA,
                                           const int *csrRowPtrA,
                                           int nnzA,
                                           const int *nnzPerRow,
                                           cuDoubleComplex *csrValC,
                                           int *csrColIndC,
                                           int *csrRowPtrC,
                                           cuDoubleComplex tol);
```

This function compresses the sparse matrix in CSR format into compressed CSR format. Given a sparse matrix A and a non-negative value threshold(in the case of complex values, only the magnitude of the real part is used in the check), the function returns a sparse matrix C, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates csrRowPtrC of m+1 elements and uses function cusparse<t>nnz_compress() to determine nnzPerRow(the number of nonzeros columns per row) and nnzC(the total number of nonzeros). Second, the user allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function cusparse<t>csr2csr_compress() is called to complete the conversion.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix A.
n	number of columns of matrix A.
descrA	the descriptor of matrix A. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) elements of matrix A.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the elements of matrix A.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
nnzA	number of nonzero elements in matrix A.
nnzPerRow	this array contains the number of elements kept in the compressed matrix, by row.
tol	on input, this contains the non-negative tolerance value used for compression. Any values in matrix A less than or equal to this value will be dropped during compression.

Output

csrValC	on output, this array contains the typed values of elements kept in the compressed matrix. Size = nnzC.
csrColIndC	on output, this integer array contains the column indices of elements kept in the compressed matrix. Size = nnzC.

csrRowPtrC	on output, this integer array contains the row pointers for elements kept in the compressed matrix. Size = m+1

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

The following is a sample code to show how to use this API.

```
#include <stdio.h>
#include <sys/time.h>
#include <cusparse.h>

#define ERR_NE(X,Y) do { if ((X) != (Y)) { \
                            fprintf(stderr,"Error in %s at %s:%d
\n",__func__,__FILE__,__LINE__); \
                            exit(-1);}} while(0)
#define CUDA_CALL(X) ERR_NE((X),cudaSuccess)
#define CUSPARSE_CALL(X) ERR_NE((X),CUSPARSE_STATUS_SUCCESS)
int main(){
    int m = 6, n = 5;
    cusparseHandle_t  handle;
    CUSPARSE_CALL( cusparseCreate(&handle) );
    cusparseMatDescr_t descrX;
    CUSPARSE_CALL(cusparseCreateMatDescr(&descrX));
    // Initialize sparse matrix
    float *X;
    CUDA_CALL(cudaMallocManaged( &X, sizeof(float) * m * n ));
    memset( X, 0, sizeof(float) * m * n );
    X[0 + 0*m] = 1.0;  X[0 + 1*m] = 3.0;
    X[1 + 1*m] = -4.0; X[1 + 2*m] = 5.0;
    X[2 + 0*m] = 2.0;  X[2 + 3*m] = 7.0;  X[2 + 4*m] = 8.0;
    X[3 + 2*m] = 6.0;  X[3 + 4*m] = 9.0;
    X[4 + 3*m] = 3.5;  X[4 + 4*m] = 5.5;
    X[5 + 0*m] = 6.5;  X[5 + 2*m] = -9.9;
    // Initialize total_nnz, and  nnzPerRowX for cusparseSdense2csr()
    int total_nnz = 13;
    int *nnzPerRowX;
    CUDA_CALL( cudaMallocManaged( &nnzPerRowX, sizeof(int) * m ));
    nnzPerRowX[0] = 2;  nnzPerRowX[1] = 2;  nnzPerRowX[2] = 3;
    nnzPerRowX[3] = 2;  nnzPerRowX[4] = 2;  nnzPerRowX[5] = 2;

    float *csrValX;
    int *csrRowPtrX;
    int *csrColIndX;
    CUDA_CALL( cudaMallocManaged( &csrValX, sizeof(float) * total_nnz) );
    CUDA_CALL( cudaMallocManaged( &csrRowPtrX, sizeof(int) * (m+1))) ;
    CUDA_CALL( cudaMallocManaged( &csrColIndX, sizeof(int) * total_nnz)) ;
```

Before calling this API, call two APIs to prepare the input.

```
/** Call cusparseSdense2csr to generate CSR format as the inputs for
    cusparseScsr2csr_compress  **/
    CUSPARSE_CALL( cusparseSdense2csr( handle, m, n, descrX, X,
                                       m, nnzPerRowX, csrValX,
                                       csrRowPtrX, csrColIndX )) ;
    float tol = 3.5;
    int *nnzPerRowY;
    int *testNNZTotal;
    CUDA_CALL (cudaMallocManaged( &nnzPerRowY, sizeof(int) * m ));
    CUDA_CALL (cudaMallocManaged( &testNNZTotal, sizeof(int)));
    memset( nnzPerRowY, 0, sizeof(int) * m );
    // cusparseSnnz_compress generates nnzPerRowY and testNNZTotal
    CUSPARSE_CALL( cusparseSnnz_compress(handle, m, descrX, csrValX,
                                         csrRowPtrX, nnzPerRowY,
                                         testNNZTotal, tol));

    float *csrValY;
    int *csrRowPtrY;
    int *csrColIndY;
    CUDA_CALL( cudaMallocManaged( &csrValY, sizeof(float) * (*testNNZTotal)));
    CUDA_CALL( cudaMallocManaged( &csrRowPtrY, sizeof(int) * (m+1)));
    CUDA_CALL( cudaMallocManaged( &csrColIndY, sizeof(int) * (*testNNZTotal)));

    CUSPARSE_CALL( cusparseScsr2csr_compress( handle, m, n, descrX, csrValX,
                                              csrColIndX, csrRowPtrX,
                                              total_nnz,  nnzPerRowY,
                                              csrValY, csrColIndY,
                                              csrRowPtrY, tol));
    /* Expect results
    nnzPerRowY:  0 2 2 2 1 2
    csrValY:     -4 5 7 8 6 9 5.5 6.5 -9.9
    csrColIndY:  1 2 3 4 2 4 4 0 2
    csrRowPtrY:  0 0 2 4 6 7 9
    */
    cudaFree(X);
    cusparseDestroy(handle);
    cudaFree(nnzPerRowX);
    cudaFree(csrValX);
    cudaFree(csrRowPtrX);
    cudaFree(csrColIndX);
    cudaFree(csrValY);
    cudaFree(nnzPerRowY);
    cudaFree(testNNZTotal);
    cudaFree(csrRowPtrY);
    cudaFree(csrColIndY);
    return 0;
}
```

12.18. cusparse<t>csr2hyb()

```
cusparseStatus_t
cusparseScsr2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const float           *csrValA,
            const int *csrRowPtrA, const int *csrColIndA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseDcsr2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const double          *csrValA,
            const int *csrRowPtrA, const int *csrColIndA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseCcsr2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const cuComplex       *csrValA,
            const int *csrRowPtrA, const int *csrColIndA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseZcsr2hyb(cusparseHandle_t handle, int m, int n,
            const cusparseMatDescr_t descrA,
            const cuDoubleComplex *csrValA,
            const int *csrRowPtrA, const int *csrColIndA,
            cusparseHybMat_t hybA, int userEllWidth,
            cusparseHybPartition_t partitionType)
```

This function converts a sparse matrix in CSR format into a sparse matrix in HYB format. It assumes that the **hybA** parameter has been initialized with **cusparseCreateHybMat()** routine before calling this function.

This function requires some amount of temporary storage and a significant amount of storage for the matrix in HYB format. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** – **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.

csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
userEllWidth	width of the regular (ELL) part of the matrix in HYB format, which should be less than maximum number of nonzeros per row and is only required if **partitionType** == **CUSPARSE_HYB_PARTITION_USER**.
partitionType	partitioning method to be used in the conversion (please refer to **cusparseHybPartition_t** for details).

Output

hybA	the matrix **A** in HYB storage format.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.19. cusparse<t>dense2csc()

```
cusparseStatus_t
cusparseSdense2csc(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const float           *A,
                   int lda, const int *nnzPerCol,
                   float           *cscValA,
                   int *cscRowIndA, int *cscColPtrA)
cusparseStatus_t
cusparseDdense2csc(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const double          *A,
                   int lda, const int *nnzPerCol,
                   double          *cscValA,
                   int *cscRowIndA, int *cscColPtrA)
cusparseStatus_t
cusparseCdense2csc(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex       *A,
                   int lda, const int *nnzPerCol,
                   cuComplex       *cscValA,
                   int *cscRowIndA, int *cscColPtrA)
cusparseStatus_t
cusparseZdense2csc(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex *A,
                   int lda, const int *nnzPerCol,
                   cuDoubleComplex *cscValA,
                   int *cscRowIndA, int *cscColPtrA)
```

This function converts the matrix **A** in dense format into a sparse matrix in CSC format. All the parameters are assumed to have been pre-allocated by the user, and the arrays are filled in based on **nnzPerCol**, which can be precomputed with **cusparse<t>nnz()**.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
A	array of dimensions **(lda, n)**.
lda	leading dimension of dense array **A**.
nnzPerCol	array of size **n** containing the number of nonzero elements per column.

Output

cscValA	<type> array of **nnz** (= **cscRowPtrA(m) - cscRowPtrA(0)**) nonzero elements of matrix **A**. It is only filled in if **copyValues** is set to **CUSPARSE_ACTION_NUMERIC**.
cscRowIndA	integer array of **nnz** (= **cscRowPtrA(m) - cscRowPtrA(0)**) row indices of the nonzero elements of matrix **A**.
cscColPtrA	integer array of **n+1** elements that contains the start of every column and the end of the last column plus one.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.20. cusparse<t>dense2csr()

```
cusparseStatus_t
cusparseSdense2csr(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const float           *A,
                   int lda, const int *nnzPerRow,
                   float           *csrValA,
                   int *csrRowPtrA, int *csrColIndA)
cusparseStatus_t
cusparseDdense2csr(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const double          *A,
                   int lda, const int *nnzPerRow,
                   double          *csrValA,
                   int *csrRowPtrA, int *csrColIndA)
cusparseStatus_t
cusparseCdense2csr(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex       *A,
                   int lda, const int *nnzPerRow,
                   cuComplex       *csrValA,
                   int *csrRowPtrA, int *csrColIndA)
cusparseStatus_t
cusparseZdense2csr(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex *A,
                   int lda, const int *nnzPerRow,
                   cuDoubleComplex *csrValA,
                   int *csrRowPtrA, int *csrColIndA)
```

This function converts the matrix **A** in dense format into a sparse matrix in CSR format. All the parameters are assumed to have been pre-allocated by the user and the arrays are filled in based on **nnzPerRow**, which can be pre-computed with **cusparse<t>nnz()**.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
A	array of dimensions **(lda, n)**.
lda	leading dimension of dense array **A**.
nnzPerRow	array of size **n** containing the number of non-zero elements per row.

Output

csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every column and the end of the last column plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the non-zero elements of matrix **A**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.21. cusparse<t>dense2hyb()

```
cusparseStatus_t
cusparseSdense2hyb(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const float           *A,
                   int lda, const int *nnzPerRow, cusparseHybMat_t hybA,
                   int userEllWidth,
                   cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseDdense2hyb(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const double          *A,
                   int lda, const int *nnzPerRow, cusparseHybMat_t
 hybA,
                   int userEllWidth,
                   cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseCdense2hyb(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex       *A,
                   int lda, const int *nnzPerRow, cusparseHybMat_t hybA,
                   int userEllWidth,
                   cusparseHybPartition_t partitionType)
cusparseStatus_t
cusparseZdense2hyb(cusparseHandle_t handle, int m, int n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex *A,
                   int lda, const int *nnzPerRow, cusparseHybMat_t hybA,
                   int userEllWidth,
                   cusparseHybPartition_t partitionType)
```

This function converts matrix **A** in dense format into a sparse matrix in HYB format. It assumes that the routine **cusparseCreateHybMat()** was used to initialize the opaque structure **hybA** and that the array **nnzPerRow** was pre-computed with **cusparse<t>nnz()**.

This function requires some amount of temporary storage and a significant amount of storage for the matrix in HYB format. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix A.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.
A	array of dimensions **(lda, n)**.
lda	leading dimension of dense array **A**.
nnzPerRow	array of size **m** containing the number of nonzero elements per row.

userEllWidth	width of the regular (ELL) part of the matrix in HYB format, which should be less than maximum number of nonzeros per row and is only required if **partitionType** == **CUSPARSE_HYB_PARTITION_USER**.
partitionType	partitioning method to be used in the conversion (please refer to **cusparseHybPartition_t** for details).

Output

hybA	the matrix **A** in HYB storage format.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.22. cusparse<t>hyb2csc()

```
cusparseStatus_t
cusparseShyb2csc(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 float       *cscValA, int *cscRowIndA, int *cscColPtrA)
cusparseStatus_t
cusparseDhyb2csc(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 double      *cscValA, int *cscRowIndA, int *cscColPtrA)

cusparseStatus_t
cusparseChyb2csc(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 cuComplex   *cscValA, int *cscRowIndA, int *cscColPtrA)
cusparseStatus_t
cusparseZhyb2csc(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 cuDoubleComplex *cscValA, int *cscRowIndA, int
 *cscColPtrA)
```

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

This function requires some amount of temporary storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
descrA	the descriptor of matrix **A** in Hyb format. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.
hybA	the matrix **A** in HYB storage format.

Output

cscValA	<type> array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) nonzero elements of matrix **A**.
cscRowIndA	integer array of **nnz** (= **cscColPtrA(m) - cscColPtrA(0)**) column indices of the non-zero elements of matrix **A**.
cscColPtrA	integer array of **m+1** elements that contains the start of every column and the end of the last row plus one.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.23. cusparse<t>hyb2csr()

```
cusparseStatus_t
cusparseShyb2csr(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 float       *csrValA, int *csrRowPtrA, int *csrColIndA)
cusparseStatus_t
cusparseDhyb2csr(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 double      *csrValA, int *csrRowPtrA, int *csrColIndA)

cusparseStatus_t
cusparseChyb2csr(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 cuComplex   *csrValA, int *csrRowPtrA, int *csrColIndA)
cusparseStatus_t
cusparseZhyb2csr(cusparseHandle_t handle,
                 const cusparseMatDescr_t descrA,
                 const cusparseHybMat_t hybA,
                 cuDoubleComplex *csrValA, int *csrRowPtrA, int
 *csrColIndA)
```

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

This function requires some amount of temporary storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
descrA	the descriptor of matrix **A** in Hyb format. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.
hybA	the matrix **A** in HYB storage format.

Output

csrValA	<type> array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every column and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m) - csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.

CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.24. cusparse<t>hyb2dense()

```
cusparseStatus_t
cusparseShyb2dense(cusparseHandle_t handle,
                   const cusparseMatDescr_t descrA,
                   const cusparseHybMat_t hybA,
                   float *A,
                   int lda)
cusparseStatus_t
cusparseDhyb2dense(cusparseHandle_t handle,
                   const cusparseMatDescr_t descrA,
                   const cusparseHybMat_t hybA,
                   double *A,
                   int lda)
cusparseStatus_t
cusparseChyb2dense(cusparseHandle_t handle,
                   const cusparseMatDescr_t descrA,
                   const cusparseHybMat_t hybA,
                   cuComplex *A,
                   int lda)
cusparseStatus_t
cusparseZhyb2dense(cusparseHandle_t handle,
                   const cusparseMatDescr_t descrA,
                   const cusparseHybMat_t hybA,
                   cuDoubleComplex *A,
                   int lda)
```

This function converts a sparse matrix in HYB format (contained in the opaque structure) into matrix **A** in dense format. The dense matrix **A** is filled in with the values of the sparse matrix and with zeros elsewhere.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
descrA	the descriptor of matrix **A** in Hyb format. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**.
hybA	the matrix **A** in HYB storage format.
lda	leading dimension of dense array **A**.

Output

A	array of dimensions **(lda, n)** that is filled in with the values of the sparse matrix.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	the internally stored hyb format parameters are invalid.
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.25. cusparse<t>nnz()

```
cusparseStatus_t
cusparseSnnz(cusparseHandle_t handle, cusparseDirection_t dirA, int m,
             int n, const cusparseMatDescr_t descrA,
             const float           *A,
             int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)
cusparseStatus_t
cusparseDnnz(cusparseHandle_t handle, cusparseDirection_t dirA, int m,
             int n, const cusparseMatDescr_t descrA,
             const double          *A,
             int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)
cusparseStatus_t
cusparseCnnz(cusparseHandle_t handle, cusparseDirection_t dirA, int m,
             int n, const cusparseMatDescr_t descrA,
             const cuComplex       *A,
             int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)
cusparseStatus_t
cusparseZnnz(cusparseHandle_t handle, cusparseDirection_t dirA, int m,
             int n, const cusparseMatDescr_t descrA,
             const cuDoubleComplex *A,
             int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)
```

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context.
dirA	direction that specifies whether to count nonzero elements by **CUSPARSE_DIRECTION_ROW** or by **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.

descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
A	array of dimensions **(lda, n)**.
lda	leading dimension of dense array **A**.

Output

nnzPerRowColumn	array of size **m** or **n** containing the number of nonzero elements per row or column, respectively.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.26. cusparseCreateIdentityPermutation()

```
cusparseStatus_t
cusparseCreateIdentityPermutation(cusparseHandle_t handle,
                                  int n,
                                  int *p);
```

This function creates an identity map. The output parameter **p** represents such map by **p = 0:1:(n-1)**.

This function is typically used with **coosort**, **csrsort**, **cscsort**, **csr2csc_indexOnly**.

Input

parameter	device or host	description
handle	**host**	handle to the cuSPARSE library context.
n	**host**	size of the map.

Output

parameter	device or host	description

p	**device**	integer array of dimensions **n**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.27. cusparseXcoosort()

```
cusparseStatus_t
cusparseXcoosort_bufferSizeExt(
                      cusparseHandle_t handle,
                      int m,
                      int n,
                      int nnz,
                      const int *cooRows,
                      const int *cooCols,
                      size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseXcoosortByRow(cusparseHandle_t handle,
                      int m,
                      int n,
                      int nnz,
                      int *cooRows,
                      int *cooCols,
                      int *P,
                      void *pBuffer);

cusparseStatus_t
cusparseXcoosortByColumn(cusparseHandle_t handle,
                         int m,
                         int n,
                         int nnz,
                         int *cooRows,
                         int *cooCols,
                         int *P,
                         void *pBuffer);
```

This function sorts COO format. The sorting is in-place. Also the user can sort by row or sort by column.

A is an **m×n** sparse matrix that is defined in COO storage format by the three arrays **cooVals**, **cooRows**, and **cooCols**.

There is no assumption for the base index of the matrix. **coosort** uses stable sort on signed integer, so the value of **cooRows** or **cooCols** can be negative.

This function **coosort()** requires buffer size returned by **coosort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **cooVal**, **P** must be set as 0:1:(nnz-1) before **coosort()**, and after **coosort()**, new sorted value array satisfies **cooVal_sorted = cooVal(P)**.

Remark: the dimension **m** and **n** are not used. If the user does not know the value of **m** or **n**, just passes a value positive. This usually happens if the user only reads a COO array first and needs to decide the dimension **m** or **n** later.

Appendix D provides a simple example of **coosort()**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
nnz	host	number of nonzero elements of matrix **A**.
cooRows	device	integer array of **nnz** unsorted row indices of **A**.
cooCols	device	integer array of **nnz** unsorted column indices of **A**.
P	device	integer array of **nnz** unsorted map indices. To construct **cooVal**, the user has to set **P=0:1:(nnz-1)**.
pBuffer	device	buffer allocated by the user; the size is returned by **coosort_bufferSizeExt()**.

Output

parameter	device or host	description
cooRows	device	integer array of **nnz** sorted row indices of **A**.
cooCols	device	integer array of **nnz** sorted column indices of **A**.
P	device	integer array of **nnz** sorted map indices.
pBufferSizeInBytes	host	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

12.28. cusparseXcsrsort()

```
cusparseStatus_t
cusparseXcsrsort_bufferSizeExt(
                 cusparseHandle_t handle,
                 int m,
                 int n,
                 int nnz,
                 const int *csrRowPtr,
                 const int *csrColInd,
                 size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseXcsrsort(cusparseHandle_t handle,
                 int m,
                 int n,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 const int *csrRowPtr,
                 int *csrColInd,
                 int *P,
                 void *pBuffer);
```

This function sorts CSR format. The stable sorting is in-place.

The matrix type is regarded as **CUSPARSE_MATRIX_TYPE_GENERAL** implicitly. In other words, any symmetric property is ignored.

This function **csrsort()** requires buffer size returned by **csrsort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **csrVal**, **P** must be set as 0:1:(nnz-1) before **csrsort()**, and after **csrsort()**, new sorted value array satisfies **csrVal_sorted = csrVal(P)**.

The general procedure is as follows:

```
// A is a 3x3 sparse matrix, base-0
//     | 1 2 3 |
// A = | 4 5 6 |
//     | 7 8 9 |
const int m = 3;
const int n = 3;
const int nnz = 9;
csrRowPtr[m+1] = { 0, 3, 6, 9}; // on device
csrColInd[nnz] = { 2, 1, 0, 0, 2,1, 1, 2, 0}; // on device
csrVal[nnz] = { 3, 2, 1, 4, 6, 5, 8, 9, 7}; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcsrsort_bufferSizeExt(handle, m, n, nnz, csrRowPtr, csrColInd,
 &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSR format
cusparseXcsrsort(handle, m, n, nnz, descrA, csrRowPtr, csrColInd, P, pBuffer);

// step 4: gather sorted csrVal
cusparseDgthr(handle, nnz, csrVal, csrVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

Input

parameter	device or host	description
handle	**host**	handle to the cuSPARSE library context.
m	**host**	number of rows of matrix **A**.
n	**host**	number of columns of matrix **A**.
nnz	**host**	number of nonzero elements of matrix **A**.
csrRowsPtr	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColInd	**device**	integer array of **nnz** unsorted column indices of **A**.
P	**device**	integer array of **nnz** unsorted map indices. To construct **csrVal**, the user has to set **P=0:1:(nnz-1)**.
pBuffer	**device**	buffer allocated by the user; the size is returned by **csrsort_bufferSizeExt()**.

Output

parameter	device or host	description
csrColInd	**device**	integer array of **nnz** sorted column indices of **A**.
P	**device**	integer array of **nnz** sorted map indices.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.29. cusparseXcscsort()

```
cusparseStatus_t
cusparseXcscsort_bufferSizeExt(
                 cusparseHandle_t handle,
                 int m,
                 int n,
                 int nnz,
                 const int *cscColPtr,
                 const int *cscRowInd,
                 size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseXcscsort(cusparseHandle_t handle,
                 int m,
                 int n,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 const int *cscColPtr,
                 int *cscRowInd,
                 int *P,
                 void *pBuffer);
```

This function sorts CSC format. The stable sorting is in-place.

The matrix type is regarded as **CUSPARSE_MATRIX_TYPE_GENERAL** implicitly. In other words, any symmetric property is ignored.

This function **cscsort()** requires buffer size returned by **cscsort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **cscVal**, **P** must be set as 0:1:(nnz-1) before **cscsort()**, and after **cscsort()**, new sorted value array satisfies **cscVal_sorted = cscVal(P)**.

The general procedure is as follows:

```
// A is a 3x3 sparse matrix, base-0
//     | 1 2  |
// A = | 4 0  |
//     | 0 8  |
const int m = 3;
const int n = 2;
const int nnz = 4;
cscColPtr[n+1] = { 0, 2, 4}; // on device
cscRowInd[nnz] = { 1, 0, 2, 0}; // on device
cscVal[nnz]    = { 4.0, 1.0, 8.0, 2.0 }; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcscsort_bufferSizeExt(handle, m, n, nnz, cscColPtr, cscRowInd,
 &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSC format
cusparseXcscsort(handle, m, n, nnz, descrA, cscColPtr, cscRowInd, P, pBuffer);

// step 4: gather sorted cscVal
cusparseDgthr(handle, nnz, cscVal, cscVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

Input

parameter	device or host	description
handle	**host**	handle to the cuSPARSE library context.
m	**host**	number of rows of matrix **A**.
n	**host**	number of columns of matrix **A**.
nnz	**host**	number of nonzero elements of matrix **A**.
cscColPtr	**device**	integer array of **n+1** elements that contains the start of every column and the end of the last column plus one.
cscRowInd	**device**	integer array of **nnz** unsorted row indices of **A**.
P	**device**	integer array of **nnz** unsorted map indices. To construct **cscVal**, the user has to set **P=0:1:(nnz-1)**.
pBuffer	**device**	buffer allocated by the user; the size is returned by **cscsort_bufferSizeExt()**.

Output

parameter	device or host	description
cscRowInd	**device**	integer array of **nnz** sorted row indices of **A**.
P	**device**	integer array of **nnz** sorted map indices.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.30. cusparseXcsru2csr()

```
cusparseStatus_t cusparseCreateCsru2csrInfo(csru2csrInfo_t *info);

cusparseStatus_t cusparseDestroyCsru2csrInfo(csru2csrInfo_t info);

cusparseStatus_t
cusparseScsru2csr_bufferSizeExt(
                           cusparseHandle_t handle,
                           int m,
                           int n,
                           int nnz,
                           float *csrVal,
                           const int *csrRowPtr,
                           int *csrColInd,
                           csru2csrInfo_t  info,
                           size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseDcsru2csr_bufferSizeExt(
                           cusparseHandle_t handle,
                           int m,
                           int n,
                           int nnz,
                           double *csrVal,
                           const int *csrRowPtr,
                           int *csrColInd,
                           csru2csrInfo_t  info,
                           size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseCcsru2csr_bufferSizeExt(
                           cusparseHandle_t handle,
                           int m,
                           int n,
                           int nnz,
                           cuComplex *csrVal,
                           const int *csrRowPtr,
                           int *csrColInd,
                           csru2csrInfo_t  info,
                           size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseZcsru2csr_bufferSizeExt(
                           cusparseHandle_t handle,
                           int m,
                           int n,
                           int nnz,
                           cuDoubleComplex *csrVal,
                           const int *csrRowPtr,
                           int *csrColInd,
                           csru2csrInfo_t  info,
                           size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseScsru2csr(cusparseHandle_t handle,
                  int m,
                  int n,
                  int nnz,
                  const cusparseMatDescr_t descrA,
                  float *csrVal,
                  const int *csrRowPtr,
                  int *csrColInd,
                  csru2csrInfo_t  info,
```

This function transfers unsorted CSR format to CSR format, and vice versa. The operation is in-place.

This function is a wrapper of **csrsort** and **gthr**. The usecase is the following scenario.

If the user has a matrix **A** of CSR format which is unsorted, and implements his own code (which can be CPU or GPU kernel) based on this special order (for example, diagonal first, then lower triangle, then upper triangle), and wants to convert it to CSR format when calling CUSPARSE library, and then convert it back when doing something else on his/her kernel. For example, suppose the user wants to solve a linear system **Ax=b** by the following iterative scheme

$$x^{(k+1)} = x^{(k)} + L^{(-1)} * (b - Ax^{(k)})$$

The code heavily uses SpMv and triangular solve. Assume that the user has an in-house design of SpMV (Sparse Matrix-Vector multiplication) based on special order of **A**. However the user wants to use CUSAPRSE library for triangular solver. Then the following code can work.

do

step 1: compute residual vector $r = b - A x^{(k)}$ by

step 2: B := sort(A), and L is lower triangular part of B
(only sort A once and keep the permutation vector)

step 3: solve $z = L^{(-1)} * (b$

step 4: add correction $x^{(k+1)} = x^{(k)} + z$

step 5: A := unsort(B)
(use permutation vector to get back the unsorted CSR)

until convergence

The requirements of step 2 and step 5 are

1. In-place operation.

2. The permutation vector **P** is hidden in an opaque structure.

3. No **cudaMalloc** inside the conversion routine. Instead, the user has to provide the buffer explicitly.

4. The conversion between unsorted CSR and sorted CSR may needs several times, but the function only generates the permutation vector **P** once.

5. The function is based on **csrsort**, **gather** and **scatter** operations.

The operation is called **csru2csr**, which means unsorted CSR to sorted CSR. Also we provide the inverse operation, called **csr2csru**.

In order to keep the permutation vector invisible, we need an opaque structure called **csru2csrInfo**. Then two functions (**cusparseCreateCsru2csrInfo**, **cusparseDestroyCsru2csrInfo**) are used to initialize and to destroy the opaque structure.

cusparse[S|D|C|Z]csru2csr_bufferSizeExt returns the size of the buffer. The permutation vector **P** is also allcated inside **csru2csrInfo**. The lifetime of the permutation vector is the same as the lifetime of **csru2csrInfo**.

cusparse[S|D|C|Z]csru2csr performs forward transformation from unsorted CSR to sorted CSR. First call uses csrsort to generate the permutation vector **P**, and subsequent call uses **P** to do transformation.

cusparse[S|D|C|Z]csr2csru performs backward transformation from sorted CSR to unsorted CSR. **P** is used to get unsorted form back.

The following tables describe parameters of **csr2csru_bufferSizeExt** and **csr2csru**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
nnz	host	number of nonzero elements of matrix **A**.
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**.
csrVal	device	<type> array of nnz unsorted nonzero elements of matrix **A**.
csrRowsPtr	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColInd	device	integer array of **nnz** unsorted column indices of **A**.
info	host	opaque structure initialized using **cusparseCreateCsru2csrInfo()**.
pBuffer	device	buffer allocated by the user; the size is returned by **csru2csr_bufferSizeExt()**.

Output

parameter	device or host	description
csrVal	device	<type> array of nnz sorted nonzero elements of matrix **A**.
csrColInd	device	integer array of **nnz** sorted column indices of **A**.
pBufferSizeInBytes	host	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n,nnz<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.31. cusparseXpruneDense2csr()

```
cusparseStatus_t
cusparseHpruneDense2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const __half *A,
    int lda,
    const __half *threshold,
    const cusparseMatDescr_t descrC,
    const __half *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseSpruneDense2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *A,
    int lda,
    const float *threshold,
    const cusparseMatDescr_t descrC,
    const float *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseDpruneDense2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *A,
    int lda,
    const double *threshold,
    const cusparseMatDescr_t descrC,
    const double *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseHpruneDense2csrNnz(
    cusparseHandle_t handle,
    int m,
    int n,
    const __half *A,
    int lda,
    const __half *threshold,
    const cusparseMatDescr_t descrC,
    int *csrRowPtrC,
    int *nnzTotalDevHostPtr,
    void *pBuffer);

cusparseStatus_t
cusparseSpruneDense2csrNnz(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *A,
    int lda,
```

This function prunes a dense matrix to a sparse matrix with CSR format.

Given a dense matrix **A** and a non-negative value **threshold**, the function returns a sparse matrix **C**, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneDense2csrNnz()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneDense2csr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneDense2csr()** by calling **pruneDense2csr_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneDense2csr()**.

Appendix E.1 provides a simple example of **pruneDense2csr()**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
A	device	array of dimension (lda, n).
lda	device	leading dimension of **A**. It must be at least max(1, m).
threshold	host or device	a value to drop the entries of A. **threshold** can point to a device memory or host memory.
descrC	host	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	device	buffer allocated by the user; the size is returned by **pruneDense2csr_bufferSizeExt()**.

Output

parameter	device or host	description
nnzTotalDevHostPtr	device or host	total number of nonzero of matrix **C**. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	device	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowsPtrC	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	device	integer array of **nnzC** column indices of **C**.
pBufferSizeInBytes	host	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n <0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.32. cusparseXpruneCsr2csr()

```
cusparseStatus_t
cusparseHpruneCsr2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const __half *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const __half *threshold,
    const cusparseMatDescr_t descrC,
    const __half *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseSpruneCsr2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const float *threshold,
    const cusparseMatDescr_t descrC,
    const float *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseDpruneCsr2csr_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const double *threshold,
    const cusparseMatDescr_t descrC,
    const double *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseHpruneCsr2csrNnz(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const __half *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const __half *threshold,
```

This function prunes a sparse matrix to a sparse matrix with CSR format.

Given a sparse matrix **A** and a non-negative value **threshold**, the function returns a sparse matrix **C**, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneCsr2csrNnz()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneCsr2csr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneCsr2csr()** by calling **pruneCsr2csr_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneCsr2csr()**.

Appendix E.2 provides a simple example of **pruneCsr2csr()**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE 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**.
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**.
csrValA	device	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowsPtrA	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 **nnzA** column indices of **A**.
threshold	host or device	a value to drop the entries of A. **threshold** can point to a device memory or host memory.
descrC	host	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	device	buffer allocated by the user; the size is returned by **pruneCsr2csr_bufferSizeExt()**.

Output

parameter	device or host	description

nnzTotalDevHostPtr	**device or host**	total number of nonzero of matrix **C**. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	**device**	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowsPtrC	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	**device**	integer array of **nnzC** column indices of **C**.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n <0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.33. cusparseXpruneDense2csrPercentage()

```
cusparseStatus_t
cusparseHpruneDense2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const __half *A,
    int lda,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const __half *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseSpruneDense2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const float *A,
    int lda,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const float *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseDpruneDense2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    const double *A,
    int lda,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const double *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseHpruneDense2csrNnzByPercentage(
    cusparseHandle_t handle,
    int m,
    int n,
    const __half *A,
    int lda,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    int *csrRowPtrC,
    int *nnzTotalDevHostPtr, /* can be on host or device */
    pruneInfo_t info,
    void *pBuffer);

cusparseStatus_t
cusparseSpruneDense2csrNnzByPercentage(
    cusparseHandle_t handle,
```

This function prunes a dense matrix to a sparse matrix by percentage.

Given a dense matrix **A** and a non-negative value **percentage**, the function computes sparse matrix **C** by the following three steps:

Step 1: sort absolute value of **A** in ascending order.

$$key := sort(|A|)$$

Step 2: choose threshold by the parameter **percentage**

```
pos = ceil(m*n*(percentage/100)) - 1
pos = min(pos, m*n-1)
pos = max(pos, 0)
threshold = key[pos]
```

Step 3: call **pruneDense2csr()** by with the parameter **threshold**.

The implementation adopts a two-step approach to do the conversion.
First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneDense2csrNnzByPercentage()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneDense2csrByPercentage()** is called to complete the conversion.

The user must obtain the size of the buffer required
by **pruneDense2csrByPercentage()** by calling
pruneDense2csrByPercentage_bufferSizeExt(), allocate the buffer, and pass the buffer pointer to **pruneDense2csrByPercentage()**.

Remark 1: the value of **percentage** must be not greater than 100. Otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Remark 2: the zeros of **A** are not ignored. All entries are sorted, including zeros. This is different from **pruneCsr2csrByPercentage()**

Appendix E.3 provides a simple example of **pruneDense2csrByPercentage()**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
A	device	array of dimension (lda, n).
lda	device	leading dimension of **A**. It must be at least max(1, m).
percentage	host	percentage <=100 and percentage >= 0
descrC	host	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.

pBuffer	**device**	buffer allocated by the user; the size is returned by **pruneDense2csrByPercentage_bufferSizeExt()**.

Output

parameter	device or host	description
nnzTotalDevHostPtr	**device or host**	total number of nonzero of matrix **C**. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	**device**	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowsPtrC	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	**device**	integer array of **nnzC** column indices of **C**.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m,n <0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.34. cusparseXpruneCsr2csrByPercentage()

```
cusparseStatus_t
cusparseHpruneCsr2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const __half *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const __half *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseSpruneCsr2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const float *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseDpruneCsr2csrByPercentage_bufferSizeExt(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    float percentage, /* between 0 to 100 */
    const cusparseMatDescr_t descrC,
    const double *csrValC,
    const int *csrRowPtrC,
    const int *csrColIndC,
    pruneInfo_t info,
    size_t *pBufferSizeInBytes);

cusparseStatus_t
cusparseHpruneCsr2csrNnzByPercentage(
    cusparseHandle_t handle,
    int m,
    int n,
    int nnzA,
    const cusparseMatDescr_t descrA,
    const __half *csrValA,
```

This function prunes a sparse matrix to a sparse matrix by percentage.

Given a sparse matrix **A** and a non-negative value **percentage**, the function computes sparse matrix **C** by the following three steps:

Step 1: sort absolute value of **A** in ascending order.

$$key := sort(|csrValA|)$$

Step 2: choose threshold by the parameter **percentage**

```
pos = ceil(nnzA*(percentage/100)) - 1
pos = min(pos, nnzA-1)
pos = max(pos, 0)
threshold = key[pos]
```

Step 3: call **pruneCsr2csr()** by with the parameter **threshold**.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneCsr2csrNnzByPercentage()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneCsr2csrByPercentage()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneCsr2csrByPercentage()** by calling **pruneCsr2csrByPercentage_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneCsr2csrByPercentage()**.

Remark 1: the value of **percentage** must be not greater than 100. Otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Appendix E.4 provides a simple example of **pruneCsr2csrByPercentage()**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE 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**.
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**.
csrValA	device	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowsPtrA	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 **nnzA** column indices of **A**.
percentage	host	percentage <=100 and percentage >= 0

descrC	host	the descriptor of matrix C. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.
pBuffer	device	buffer allocated by the user; the size is returned by pruneCsr2csrByPercentage_bufferSizeExt().

Output

parameter	device or host	description
nnzTotalDevHostPtr	device or host	total number of nonzero of matrix C. nnzTotalDevHostPtr can point to a device memory or host memory.
csrValC	device	<type> array of nnzC nonzero elements of matrix C.
csrRowsPtrC	device	integer array of m+1 elements that contains the start of every row and the end of the last row plus one.
csrColIndC	device	integer array of nnzC column indices of C.
pBufferSizeInBytes	host	number of bytes of the buffer.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (m,n <0).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

12.35. cusparse<t>nnz_compress()

```
cusparseStatus_t
cusparseSnnz_compress(cusparseHandle_t handle,
                      int m,
                      const cusparseMatDescr_t descr,
                      const float *csrValA,
                      const int * csrRowPtrA,
                      int *nnzPerRow,
                      int *nnzC,
                      float tol)
cusparseStatus_t
cusparseDnnz_compress(cusparseHandle_t handle,
                      int m,
                      const cusparseMatDescr_t descr,
                      const double *csrValA,
                      const int * csrRowPtrA,
                      int *nnzPerRow,
                      int *nnzC,
                      double tol)
cusparseStatus_t
cusparseCnnz_compress(cusparseHandle_t handle,
                      int m,
                      const cusparseMatDescr_t descr,
                      const cuComplex *csrValA,
                      const int * csrRowPtrA,
                      int *nnzPerRow,
                      int *nnzC,
                      cuComplex tol)
cusparseStatus_t
cusparseZnnz_compress(cusparseHandle_t handle,
                      int m,
                      const cusparseMatDescr_t descr,
                      const cuDoubleComplex *csrValA,
                      const int * csrRowPtrA,
                      int *nnzPerRow,
                      int *nnzC,
                      cuDoubleComplex tol)
```

This function is the step one to convert from csr format to compressed csr format.

Given a sparse matrix A and a non-negative value threshold, the function returns nnzPerRow(the number of nonzeros columns per row) and nnzC(the total number of nonzeros) of a sparse matrix C, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

A key assumption for the cuComplex and cuDoubleComplex case is that this tolerance is given as the real part. For example tol = 1e-8 + 0*i and we extract cureal, that is the x component of this struct.

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.

descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	csr noncompressed values array
csrRowPtrA	the corresponding input noncompressed row pointer.
tol	non-negative tolerance to determine if a number less than or equal to it.

Output

nnzPerRow	this array contains the number of elements whose absolute values are greater than tol per row.
nnzC	host/device pointer of the total number of elements whose absolute values are greater than tol.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**m, n<0**).
CUSPARSE_STATUS_ARCH_MISMATCH	the device does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	the function failed to launch on the GPU.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

Chapter 13. CUSPARSE GENERIC API REFERENCE

Starting with CUDA 10.1, the cuSPARSE Library provides a new set of API to compute sparse matrix-matrix multiplication (SpMM), in addition to the existing legacy API. All following functions and types operate on matrix problems of the form:

$$C = \alpha AB + \beta C$$

where A is sparse matrix, B and C are dense matrices. They are considered a *preview feature* i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases.

13.1. Generic Types Reference

The cuSPARSE generic type references are described in this section.

13.1.1. cudaDataType_t

The section describes types shared by multiple CUDA Libraries and defined in the header file **`library_types.h`**. The **`cudaDataType_t`** type is an enumerant to specify the data precision. It is used when the data reference does not carry the type itself (e.g void *) For example, it is used in the routine **`cusparseSpMM()`**

Value	Meaning
`CUDA_R_16F`	the data type is 16-bit floating-point (**`__half`**)
`CUDA_C_16F`	the data type is 16-bit complex floating-point (**`__half2`**)
`CUDA_R_32F`	the data type is 32-bit floating-point (**`float`**)
`CUDA_C_32F`	the data type is 32-bit complex floating-point (**`cuComplex`**)
`CUDA_R_64F`	the data type is 64-bit floating-point (**`double`**)
`CUDA_C_64F`	the data type is 64-bit complex floating-point (**`cuDoubleComplex`**)

Value	Meaning
`CUDA_R_8I`	the data type is 8-bit signed integer (`int8_t`)

13.1.2. cusparseFormat_t

This type indicates the format of the sparse matrix. Currently, only COO format is supported.

Value	Meaning
`CUSPARSE_FORMAT_COO`	the matrix is stored in Coordinate (COO) format.

13.1.3. cusparseOrder_t

This type indicates the memory layout of a dense matrix. Currently, only column-major layout is supported.

Value	Meaning
`CUSPARSE_ORDER_COL`	the matrix is stored in column-major.

13.1.4. cusparseSpMMAlg_t

This type indicates the algorithm parameter used for **`cusparseSpMM()`** and **`cusparseSpMM_bufferSize()`** sparse matrix-matrix multiplication.

Value	Meaning
`CUSPARSE_COOMM_ALG1`	COO matrix-matrix multiplication specialized for small matrices. The algorithm provides non-deterministic results (different execution could provide slighlty different output). It supports batched matrix computation
`CUSPARSE_COOMM_ALG2`	COO matrix-matrix multiplication specialized for small matrices. The algorithm provides deterministic results (generate the same bit-wise output). It supports only op(A) CUSPARSE_OPERATION_NON_TRANSPOSE
`CUSPARSE_COOMM_ALG3`	COO matrix-matrix multiplication specialized for large matrices. The algorithm provides non-deterministic results (different execution could provide slighlty different output). It does not support cuDoubleComplex data type

13.1.5. cusparseIndexType_t

This type indicates the index type for rappresenting the sparse matrix indices.

Value	Meaning
CUSPARSE_INDEX_16U	16-bit unsigned integer for both row and column COO indices [1, 65535]
CUSPARSE_INDEX_32I	32-bit signed integer for for both row and column COO indices [1, 2147483647]

13.2. Generic Sparse API helper functions

The cuSPARSE helper functions for sparse matrix descriptor are described in this section.

13.2.1. cusparseCreateCoo()

```
cusparseStatus_t
cusparseCreateCoo(cusparseSpMatDescr_t* spMatDescr,
                  int                   rows,
                  int                   cols,
                  int                   nnz,
                  void*                 cooRowInd,
                  void*                 cooColInd,
                  void*                 cooValues,
                  cusparseIndexType_t   cooIdxType,
                  cusparseIndexBase_t   idxBase,
                  cudaDataType          valueType)
```

This function initializes the sparse matrix descriptor **spMatDescr** in the COO format.

Input

rows	number of rows of the sparse matrix.
cols	number of cols of sparse matrix.
nnz	number of nonzero elements of the sparse matrix.
cooRowInd	integer array of **nnz** uncompressed row indices.
cooColInd	integer array of **nnz** uncompressed column indices.
cooValues	value array of size **nnz** of the sparse matrix.
cooIdxType	Index base **CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.
idxBase	Index data type **CUSPARSE_INDEX_16U** or **CUSPARSE_INDEX_32I**.
valueType	value type.

Output

spMatDescr	sparse matrix descriptor.

13.2.2. cusparseDestroySpMat()

```
cusparseStatus_t
cusparseDestroySpMat(cusparseSpMatDescr_t spMatDescr)
```

This function releases the memory allocated for the sparse matrix descriptor **spMatDescr**.

Input

spMatDescr	sparse matrix descriptor.

13.2.3. cusparseSpMatGetNumBatches()

```
cusparseStatus_t
cusparseSpMatGetNumBatches(const cusparseSpMatDescr_t spMatDescr,
                           int*                       batchCount)
```

This function returns the number of batches of the sparse matrix descriptor **spMatDescr**.

Input

spMatDescr	sparse matrix descriptor.

Output

batchCount	number of batches of the sparse matrix.

13.2.4. cusparseCooGet()

```
cusparseStatus_t
cusparseCooGet(const cusparseSpMatDescr_t spMatDescr,
               int*                       rows,
               int*                       cols,
               int*                       nnz,
               void**                     cooRowInd,
               void**                     cooColInd,
               void**                     cooValues,
               cusparseIndexType_t*       idxType,
               cusparseIndexBase_t*       idxBase,
               cudaDataType*              valueType)
```

This function returns the **spMatDescr** fields of the sparse matrix descriptor.

Input

spMatDescr	sparse matrix descriptor.

Output

rows	number of rows of the sparse matrix.
cols	number of cols of sparse matrix.
nnz	number of nonzero elements of the sparse matrix.

cooRowInd	integer array of nnz uncompressed row indices.
cooColInd	integer array of nnz uncompressed column indices.
cooIdxType	Index base **CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.
idxBase	Index data type **CUSPARSE_INDEX_16U** or **CUSPARSE_INDEX_32I**.

13.2.5. cusparseCooGet()

```
cusparseStatus_t
cusparseSpMatGetFormat(const cusparseSpMatDescr_t spMatDescr,
                       cusparseFormat_t*          format)
```

This function returns the format of the sparse matrix descriptor **spMatDescr**.

Input

spMatDescr	sparse matrix descriptor.

Output

format	storage format of the sparse matrix.

13.2.6. cusparseSpMatGetIndexBase()

```
cusparseStatus_t
cusparseSpMatGetIndexBase(const cusparseSpMatDescr_t spMatDescr,
                          cusparseIndexBase_t*       idxBase)
```

This function returns the index base of the sparse matrix descriptor spMatDescr.

Input

spMatDescr	sparse matrix descriptor.

Output

idxBase	index base of the sparse matrix.

13.2.7. cusparseSpMatSetNumBatches()

```
cusparseStatus_t
cusparseSpMatSetNumBatches(cusparseSpMatDescr_t spMatDescr,
                           int                  batchCount)
```

This function sets the number of batches of the sparse matrix descriptor **spMatDescr**.

Input

spMatDescr	sparse matrix descriptor.
batchCount	number of batches for the sparse matrix descriptor.

13.3. Generic Dense API helper functions

The cuSPARSE helper functions for dense matrix descriptor are described in this section.

13.3.1. cusparseCreateDnMat()

```
cusparseStatus_t
cusparseCreateDnMat(cusparseDnMatDescr_t* dnMatDescr,
                    size_t                rows,
                    size_t                cols,
                    int64_t               ld,
                    void*                 valuesPtr,
                    cudaDataType          type,
                    cusparseOrder_t       order)
```

This function initializes the dense matrix descriptor **dnMatDescr**.

Input

rows	number of rows of the dense matrix.
cols	number of cols of the dense matrix.
ld	leading dimension of the dense matrix.
valuesPtr	value array of dimensions **(ld x cols)**.
type	datatype of the dense matrix.
order	memory layout **CUSPARSE_ORDER_COL**.

Output

dnMatDescr	dense matrix descriptor.

13.3.2. cusparseDestroyDnMat()

```
cusparseStatus_t
cusparseDestroySpMat(cusparseDnMatDescr_t dnMatDescr)
```

This function releases the memory allocated for the sparse matrix descriptor **dnMatDescr**.

Input

dnMatDescr	dense matrix descriptor.

13.3.3. cusparseDnMatGet()

```
cusparseStatus_t
cusparseDnMatGet(const cusparseDnMatDescr_t dnMatDescr,
                 size_t*                    rows,
                 size_t*                    cols,
                 int64_t*                   ld,
                 void**                     valuesPtr,
                 cudaDataType*              type,
                 cusparseOrder_t*           order)
```

This function returns the **dnMatDescr** fields of the dense matrix descriptor.

Input

dnMatDescr	dense matrix descriptor.

Output

rows	number of rows of the dense matrix.
cols	number of cols of dense matrix.
ld	leading dimension of the dense matrix.
valuesPtr	value array of dimensions **(ld x cols)**.
type	datatype of the dense matrix.
order	memory layout **CUSPARSE_ORDER_COL**.

13.3.4. cusparseDnMatGetStridedBatch()

```
cusparseStatus_t
cusparseDnMatGetStridedBatch(const cusparseDnMatDescr_t dnMatDescr,
                             int*                       batchCount,
                             size_t*                    batchStride)
```

This function returns the number of batches of the dense matrix descriptor **dnMatDescr**.

Input

dnMatDescr	dense matrix descriptor.

Output

batchCount	number of batches of the dense matrix.

13.3.5. cusparseDnMatSetNumBatches()

```
cusparseStatus_t
cusparseDnMatSetStridedBatch(cusparseDnMatDescr_t dnMatDescr,
                             int                  batchCount,
                             size_t               batchStride)
```

This function sets the number of batches of the dense matrix descriptor **dnMatDescr**.

Input

dnMatDescr	dense matrix descriptor.
batchCount	number of batches for the dense matrix descriptor.
batchStride	address offset between a matrix and the next one.

13.4. Generic SpMM API functions

The sparse matrix-matrix multiplication functions are described in this section.

13.4.1. cusparseSpMM()

```
cusparseStatus_t
cusparseSpMM(cusparseHandle_t           handle,
             cusparseOperation_t        transA,
             cusparseOperation_t        transB,
             const void*                alpha,
             const cusparseSpMatDescr_t matA,
             const cusparseDnMatDescr_t matB,
             const void*                beta,
             cusparseDnMatDescr_t       matC,
             cudaDataType               computeType,
             cusparseSpMMAlg_t          alg,
             void*                      externalBuffer)
```

This function performs one of the following sparse matrix-matrix operations:

$$C = \alpha op(A) op(B) + \beta C$$

where op(A) is a sparse matrix with dimensions $m \times k$ and B and C are matrices stored in column-major format with dimensions of op(B): $k \times n$ and the dimensions of C: $m \times n$.

$$\text{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

Input

handle	handle to the cuSPARSE library context..
transA	the operation of op(**A**).
transB	the operation of op(**B**).
alpha	scaling factor for A*B; of same type as computeType.
matA	sparse matrix descriptor of the matrix **A**.
matB	dense matrix descriptor of the matrix **B**.
beta	scaling factor for C; of same type as computeType.
matC	dense matrix descriptor of the matrix **C**.
computeType	computation type.

alg	algorithm implementation for SpMM. see cusparseSpMMAlg_t for possible values.
externalBuffer	pointer to workspace buffer.

Output

matC	updated dense matrix descriptor of the matrix C.

13.4.2. cusparseSpMM_bufferSize()

```
cusparseStatus_t
cusparseSpMM_bufferSize(cusparseHandle_t           handle,
                        cusparseOperation_t        transA,
                        cusparseOperation_t        transB,
                        const void*                alpha,
                        const cusparseSpMatDescr_t matA,
                        const cusparseDnMatDescr_t matB,
                        const void*                beta,
                        cusparseDnMatDescr_t       matC,
                        cudaDataType               computeType,
                        cusparseSpMMAlg_t          alg,
                        size_t*                    bufferSize)
```

This function returns the size of the workspace needed by **cusparseSpMM()**. User needs to allocate a buffer of this size and give that buffer to **cusparseSpMM()** as an argument. All the arguments are similar to **cusparseSpMM()** except the following output argument.

Output

bufferSize	number of bytes of workspace needed by **cusparseSpMM()**.

Chapter 14. APPENDIX A: CUSPARSE LIBRARY C++ EXAMPLE

For sample code reference please see the example code below. It shows an application written in C++ using the cuSPARSE library API. The code performs the following actions:

1. Creates a sparse test matrix in COO format.
2. Creates a sparse and dense vector.
3. Allocates GPU memory and copies the matrix and vectors into it.
4. Initializes the cuSPARSE library.
5. Creates and sets up the matrix descriptor.
6. Converts the matrix from COO to CSR format.
7. Exercises Level 1 routines.
8. Exercises Level 2 routines.
9. Exercises Level 3 routines.
10. Destroys the matrix descriptor.

11. Releases resources allocated for the cuSPARSE library.

```
//Example: Application using C++ and the CUSPARSE library
//---------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cusparse.h"

#define CLEANUP(s)                                      \
do {                                                    \
    printf ("%s\n", s);                                 \
    if (yHostPtr)           free(yHostPtr);             \
    if (zHostPtr)           free(zHostPtr);             \
    if (xIndHostPtr)        free(xIndHostPtr);          \
    if (xValHostPtr)        free(xValHostPtr);          \
    if (cooRowIndexHostPtr) free(cooRowIndexHostPtr);\
    if (cooColIndexHostPtr) free(cooColIndexHostPtr);\
    if (cooValHostPtr)      free(cooValHostPtr);        \
    if (y)                  cudaFree(y);                \
    if (z)                  cudaFree(z);                \
    if (xInd)               cudaFree(xInd);             \
    if (xVal)               cudaFree(xVal);             \
    if (csrRowPtr)          cudaFree(csrRowPtr);        \
    if (cooRowIndex)        cudaFree(cooRowIndex);      \
    if (cooColIndex)        cudaFree(cooColIndex);      \
    if (cooVal)             cudaFree(cooVal);           \
    if (descr)              cusparseDestroyMatDescr(descr);\
    if (handle)             cusparseDestroy(handle); \
    cudaDeviceReset();          \
    fflush (stdout);                                    \
} while (0)

int main(){
    cudaError_t cudaStat1,cudaStat2,cudaStat3,cudaStat4,cudaStat5,cudaStat6;
    cusparseStatus_t status;
    cusparseHandle_t handle=0;
    cusparseMatDescr_t descr=0;
    int *    cooRowIndexHostPtr=0;
    int *    cooColIndexHostPtr=0;
    double * cooValHostPtr=0;
    int *    cooRowIndex=0;
    int *    cooColIndex=0;
    double * cooVal=0;
    int *    xIndHostPtr=0;
    double * xValHostPtr=0;
    double * yHostPtr=0;
    int *    xInd=0;
    double * xVal=0;
    double * y=0;
    int *    csrRowPtr=0;
    double * zHostPtr=0;
    double * z=0;
    int      n, nnz, nnz_vector;
    double dzero =0.0;
    double dtwo  =2.0;
    double dthree=3.0;
    double dfive =5.0;

    printf("testing example\n");
    /* create the following sparse test matrix in COO format */
    /* |1.0     2.0 3.0|
       |    4.0        |
       |5.0     6.0 7.0|
       |    8.0     9.0| */
    n=4; nnz=9;
    cooRowIndexHostPtr = (int *)   malloc(nnz*sizeof(cooRowIndexHostPtr[0]));
    cooColIndexHostPtr = (int *)   malloc(nnz*sizeof(cooColIndexHostPtr[0]));
    cooValHostPtr      = (double *)malloc(nnz*sizeof(cooValHostPtr[0]));
    if ((!cooRowIndexHostPtr) || (!cooColIndexHostPtr) || (!cooValHostPtr)){
        CLEANUP("Host malloc failed (matrix)");
        return 1;
    }
    cooRowIndexHostPtr[0]=0; cooColIndexHostPtr[0]=0; cooValHostPtr[0]=1.0;
    cooRowIndexHostPtr[1]=0; cooColIndexHostPtr[1]=2; cooValHostPtr[1]=2.0;
    cooRowIndexHostPtr[2]=0; cooColIndexHostPtr[2]=3; cooValHostPtr[2]=3.0;
```

Chapter 15. APPENDIX B: CUSPARSE FORTRAN BINDINGS

The cuSPARSE library is implemented using the C-based CUDA toolchain, and it thus provides a C-style API that makes interfacing to applications written in C or C++ trivial. There are also many applications implemented in Fortran that would benefit from using cuSPARSE, and therefore a cuSPARSE Fortran interface has been developed.

Unfortunately, Fortran-to-C calling conventions are not standardized and differ by platform and toolchain. In particular, differences may exist in the following areas:

Symbol names (capitalization, name decoration)

Argument passing (by value or reference)

Passing of pointer arguments (size of the pointer)

To provide maximum flexibility in addressing those differences, the cuSPARSE Fortran interface is provided in the form of wrapper functions, which are written in C and are located in the file **`cusparse_fortran.c`**. This file also contains a few additional wrapper functions (for **`cudaMalloc()`**, **`cudaMemset`**, and so on) that can be used to allocate memory on the GPU.

The cuSPARSE Fortran wrapper code is provided as an example only and needs to be compiled into an application for it to call the cuSPARSE API functions. Providing this source code allows users to make any changes necessary for a particular platform and toolchain.

The cuSPARSE Fortran wrapper code has been used to demonstrate interoperability with the compilers g95 0.91 (on 32-bit and 64-bit Linux) and g95 0.92 (on 32-bit and 64-bit Mac OS X). In order to use other compilers, users have to make any changes to the wrapper code that may be required.

The direct wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all cuSPARSE functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPU memory space (using **`CUDA_MALLOC()`** and **`CUDA_FREE()`**) and to copy data between GPU and CPU memory spaces (using the **`CUDA_MEMCPY()`** routines). The sample wrappers provided in **`cusparse_fortran.c`** map device pointers to the OS-

dependent type **size_t**, which is 32 bits wide on 32-bit platforms and 64 bits wide on a 64-bit platforms.

One approach to dealing with index arithmetic on device pointers in Fortran code is to use C-style macros and to use the C preprocessor to expand them. On Linux and Mac OS X, preprocessing can be done by using the option **'-cpp'** with g95 or gfortran. The function **GET_SHIFTED_ADDRESS()**, provided with the cuSPARSE Fortran wrappers, can also be used, as shown in example B.

Example B shows the the C++ of example A implemented in Fortran 77 on the host. This example should be compiled with **ARCH_64** defined as 1 on a 64-bit OS system and as undefined on a 32-bit OS system. For example, on g95 or gfortran, it can be done directly on the command line using the option **-cpp -DARCH_64=1**.

15.1. Example B, Fortran Application

```
c     #define ARCH_64 0
c     #define ARCH_64 1

      program cusparse_fortran_example
      implicit none
      integer cuda_malloc
      external cuda_free
      integer cuda_memcpy_c2fort_int
      integer cuda_memcpy_c2fort_real
      integer cuda_memcpy_fort2c_int
      integer cuda_memcpy_fort2c_real
      integer cuda_memset
      integer cusparse_create
      external cusparse_destroy
      integer cusparse_get_version
      integer cusparse_create_mat_descr
      external cusparse_destroy_mat_descr
      integer cusparse_set_mat_type
      integer cusparse_get_mat_type
      integer cusparse_get_mat_fill_mode
      integer cusparse_get_mat_diag_type
      integer cusparse_set_mat_index_base
      integer cusparse_get_mat_index_base
      integer cusparse_xcoo2csr
      integer cusparse_dsctr
      integer cusparse_dcsrmv
      integer cusparse_dcsrmm
      external get_shifted_address
#if ARCH_64
      integer*8 handle
      integer*8 descrA
      integer*8 cooRowIndex
      integer*8 cooColIndex
      integer*8 cooVal
      integer*8 xInd
      integer*8 xVal
      integer*8 y
      integer*8 z
      integer*8 csrRowPtr
      integer*8 ynp1
#else
      integer*4 handle
      integer*4 descrA
      integer*4 cooRowIndex
      integer*4 cooColIndex
      integer*4 cooVal
      integer*4 xInd
      integer*4 xVal
      integer*4 y
      integer*4 z
      integer*4 csrRowPtr
      integer*4 ynp1
#endif
      integer status
      integer cudaStat1,cudaStat2,cudaStat3
      integer cudaStat4,cudaStat5,cudaStat6
      integer n, nnz, nnz_vector
      parameter (n=4, nnz=9, nnz_vector=3)
      integer cooRowIndexHostPtr(nnz)
      integer cooColIndexHostPtr(nnz)
      real*8  cooValHostPtr(nnz)
      integer xIndHostPtr(nnz_vector)
      real*8  xValHostPtr(nnz_vector)
      real*8  yHostPtr(2*n)
      real*8  zHostPtr(2*(n+1))
      integer i, j
      integer version, mtype, fmode, dtype, ibase
      real*8  dzero,dtwo,dthree,dfive
      real*8  epsilon


      write(*,*) "testing fortran example"
```

Chapter 16. APPENDIX C: EXAMPLES OF SPARSE MATRIX VECTOR MULTIPLICATION

16.1. power method

This chapter provides a simple example in the C programming language of eigenvalue problem

$$A*x = \lambda*x$$

A is a 4x4 sparse matrix,

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

The goal is to find the largest eigen-pair by power method. The following code uses **`csrmv_mp`** inside the loop of power method.

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include csrmvmp_example.cpp
 *   g++ -fopenmp -o csrmvmp_example csrmvmp_example.o -L/usr/local/cuda/lib64 -
lcublas -lcusparse -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <cusparse.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[])
{
    cublasHandle_t cublasH = NULL;
    cusparseHandle_t cusparseH = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;

    cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
    cusparseStatus_t cusparseStat = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int n = 4;
    const int nnzA = 9;
/*
 *       |    1     0     2     3   |
 *       |    0     4     0     0   |
 *   A = |    5     0     6     7   |
 *       |    0     8     0     9   |
 *
 * eigevales are { -0.5311, 7.5311, 9.0000, 4.0000 }
 *
 * The largest eigenvaluse is 9 and corresponding eigenvector is
 *
 *       | 0.3029  |
 *   v = |     0   |
 *       | 0.9350  |
 *       | 0.1844  |
 */
```

prepare matrix A

```
    const int csrRowPtrA[n+1] = { 0, 3, 4, 7, 9 };
    const int csrColIndA[nnzA] = {0, 2, 3, 1, 0, 2, 3, 1, 3 };
    const double csrValA[nnzA] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 };
    const double lambda_exact[n] = { 9.0000, 7.5311, 4.0000, -0.5311 };
    const double x0[n] = {1.0, 2.0, 3.0, 4.0 }; /* initial guess */
    double x[n]; /* numerical eigenvector */

    int *d_csrRowPtrA = NULL;
    int *d_csrColIndA = NULL;
    double *d_csrValA = NULL;

    double *d_x = NULL; /* eigenvector */
    double *d_y = NULL; /* workspace */

    const double tol = 1.e-6;
    const int max_ites = 30;

    const double h_one  = 1.0;
    const double h_zero = 0.0;

    printf("example of csrmv_mp \n");
    printf("tol = %E \n", tol);
    printf("max. iterations = %d \n", max_ites);

    printf("1st eigenvaluse is %f\n", lambda_exact[0] );
    printf("2nd eigenvaluse is %f\n", lambda_exact[1] );

    double alpha = lambda_exact[1]/lambda_exact[0] ;
    printf("convergence rate is %f\n", alpha );

    double est_iterations = log(tol)/log(alpha);
    printf("# of iterations required is %d\n", (int)ceil(est_iterations)  );

/* step 1: create cublas/cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    cublasStat = cublasCreate(&cublasH);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    cublasStat = cublasSetStream(cublasH, stream);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    cusparseStat = cusparseCreate(&cusparseH);
    assert(CUSPARSE_STATUS_SUCCESS == cusparseStat);

    cusparseStat = cusparseSetStream(cusparseH, stream);
    assert(CUSPARSE_STATUS_SUCCESS == cusparseStat);

/* step 2: configuration of matrix A */
    cusparseStat = cusparseCreateMatDescr(&descrA);
    assert(CUSPARSE_STATUS_SUCCESS == cusparseStat);

    cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ZERO);
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );
```

power method

```
/* step 3: copy A and x0 to device */
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int) * (n+1) );
    cudaStat2 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int) * nnzA );
    cudaStat3 = cudaMalloc ((void**)&d_csrValA   , sizeof(double) * nnzA );
    cudaStat4 = cudaMalloc ((void**)&d_x         , sizeof(double) * n );
    cudaStat5 = cudaMalloc ((void**)&d_y         , sizeof(double) * n );
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);
    assert(cudaSuccess == cudaStat4);
    assert(cudaSuccess == cudaStat5);

    cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int) * (n+1)   ,
 cudaMemcpyHostToDevice);
    cudaStat2 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int) * nnzA    ,
 cudaMemcpyHostToDevice);
    cudaStat3 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(double) * nnzA ,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);


/*
 * step 4: power method
 */
    double lambda = 0.0;
    double lambda_next = 0.0;

/*
 *  4.1: initial guess x0
 */
    cudaStat1 = cudaMemcpy(d_x, x0, sizeof(double) * n, cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

    for(int ite = 0 ; ite < max_ites ; ite++ ){
/*
 *  4.2: normalize vector x
 *       x = x / |x|
 */
        double nrm2_x;
        cublasStat = cublasDnrm2_v2(cublasH,
                                    n,
                                    d_x,
                                    1, // incx,
                                    &nrm2_x  /* host pointer */
                                   );
        assert(CUBLAS_STATUS_SUCCESS == cublasStat);

        double one_over_nrm2_x = 1.0 / nrm2_x;
        cublasStat = cublasDscal_v2( cublasH,
                                     n,
                                     &one_over_nrm2_x,  /* host pointer */
                                     d_x,
                                     1 // incx
                                    );
        assert(CUBLAS_STATUS_SUCCESS == cublasStat);
```

...

```
/*
 *  4.3: y = A*x
 */
        cusparseStat = cusparseDcsrmv_mp(cusparseH,
                                         CUSPARSE_OPERATION_NON_TRANSPOSE,
                                         n,
                                         n,
                                         nnzA,
                                         &h_one,
                                         descrA,
                                         d_csrValA,
                                         d_csrRowPtrA,
                                         d_csrColIndA,
                                         d_x,
                                         &h_zero,
                                         d_y);
        assert(CUSPARSE_STATUS_SUCCESS == cusparseStat);

/*
 *  4.4: lambda = y**T*x
 */
        cublasStat = cublasDdot_v2 ( cublasH,
                                     n,
                                     d_x,
                                     1, // incx,
                                     d_y,
                                     1, // incy,
                                     &lambda_next  /* host pointer */
                                   );
        assert(CUBLAS_STATUS_SUCCESS == cublasStat);

        double lambda_err = fabs( lambda_next - lambda_exact[0] );
        printf("ite %d: lambda = %f, error = %E\n", ite, lambda_next,
 lambda_err );
/*
 *  4.5: check if converges
 */
        if ( (ite > 0) &&
             fabs( lambda - lambda_next ) < tol
        ){
            break; // converges
        }

/*
 *  4.6: x := y
 *       lambda = lambda_next
 *
 *  so new approximation is (lambda, x), x is not normalized.
 */
        cudaStat1 = cudaMemcpy(d_x, d_y, sizeof(double) * n ,
 cudaMemcpyDeviceToDevice);
        assert(cudaSuccess == cudaStat1);

        lambda = lambda_next;
    }
```

report largest eigenvalue and eigenvector.

```
/*
 * step 5: report eigen-pair
 */
    cudaStat1 = cudaMemcpy(x, d_x, sizeof(double) * n, cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);

    printf("largest eigenvalue is %E\n", lambda );

    printf("eigenvector = (matlab base-1)\n");
    printMatrix(n, 1, x, n, "V0");
    printf("=====\n");


/* free resources */
    if (d_csrRowPtrA  ) cudaFree(d_csrRowPtrA);
    if (d_csrColIndA  ) cudaFree(d_csrColIndA);
    if (d_csrValA     ) cudaFree(d_csrValA);
    if (d_x           ) cudaFree(d_x);
    if (d_y           ) cudaFree(d_y);

    if (cublasH       ) cublasDestroy(cublasH);
    if (cusparseH     ) cusparseDestroy(cusparseH);
    if (stream        ) cudaStreamDestroy(stream);
    if (descrA        ) cusparseDestroyMatDescr(descrA);

    cudaDeviceReset();

    return 0;
}
```

Chapter 17. APPENDIX D: EXAMPLES OF SORTING

17.1. COO sort

This chapter provides a simple example in the C programming language of sorting of COO format.

A is a 3x3 sparse matrix,

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 \\ 0.0 & 5.0 & 0.0 \\ 0.0 & 8.0 & 0.0 \end{pmatrix}$$

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include coosort.cpp
 *   g++ -o coosort.cpp coosort.o -L/usr/local/cuda/lib64 -lcusparse -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    cudaError_t cudaStat6 = cudaSuccess;

/*
 * A is a 3x3 sparse matrix
 *     | 1 2 0 |
 * A = | 0 5 0 |
 *     | 0 8 0 |
 */
    const int m = 3;
    const int n = 3;
    const int nnz = 4;

#if 0
/* index starts at 0 */
    int h_cooRows[nnz] = {2, 1, 0, 0 };
    int h_cooCols[nnz] = {1, 1, 0, 1 };
#else
/* index starts at -2 */
    int h_cooRows[nnz] = {0, -1, -2, -2 };
    int h_cooCols[nnz] = {-1, -1, -2, -1 };
#endif
    double h_cooVals[nnz] = {8.0, 5.0, 1.0, 2.0 };
    int h_P[nnz];

    int *d_cooRows = NULL;
    int *d_cooCols = NULL;
    int *d_P       = NULL;
    double *d_cooVals = NULL;
    double *d_cooVals_sorted = NULL;
    size_t pBufferSizeInBytes = 0;
    void *pBuffer = NULL;

    printf("m = %d, n = %d, nnz=%d \n", m, n, nnz );
```

...

```
/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: allocate buffer */
    status = cusparseXcoosort_bufferSizeExt(
        handle,
        m,
        n,
        nnz,
        d_cooRows,
        d_cooCols,
        &pBufferSizeInBytes
    );
    assert( CUSPARSE_STATUS_SUCCESS == status);

    printf("pBufferSizeInBytes = %lld bytes \n", (long long)pBufferSizeInBytes);

    cudaStat1 = cudaMalloc( &d_cooRows, sizeof(int)*nnz);
    cudaStat2 = cudaMalloc( &d_cooCols, sizeof(int)*nnz);
    cudaStat3 = cudaMalloc( &d_P      , sizeof(int)*nnz);
    cudaStat4 = cudaMalloc( &d_cooVals, sizeof(double)*nnz);
    cudaStat5 = cudaMalloc( &d_cooVals_sorted, sizeof(double)*nnz);
    cudaStat6 = cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

    assert( cudaSuccess == cudaStat1 );
    assert( cudaSuccess == cudaStat2 );
    assert( cudaSuccess == cudaStat3 );
    assert( cudaSuccess == cudaStat4 );
    assert( cudaSuccess == cudaStat5 );
    assert( cudaSuccess == cudaStat6 );

    cudaStat1 = cudaMemcpy(d_cooRows, h_cooRows, sizeof(int)*nnz   ,
 cudaMemcpyHostToDevice);
    cudaStat2 = cudaMemcpy(d_cooCols, h_cooCols, sizeof(int)*nnz   ,
 cudaMemcpyHostToDevice);
    cudaStat3 = cudaMemcpy(d_cooVals, h_cooVals, sizeof(double)*nnz,
 cudaMemcpyHostToDevice);
    cudaStat4 = cudaDeviceSynchronize();
    assert( cudaSuccess == cudaStat1 );
    assert( cudaSuccess == cudaStat2 );
    assert( cudaSuccess == cudaStat3 );
    assert( cudaSuccess == cudaStat4 );

/* step 3: setup permutation vector P to identity */
    status = cusparseCreateIdentityPermutation(
        handle,
        nnz,
        d_P);
    assert( CUSPARSE_STATUS_SUCCESS == status);
```

...

```
/* step 4: sort COO format by Row */
    status = cusparseXcoosortByRow(
        handle,
        m,
        n,
        nnz,
        d_cooRows,
        d_cooCols,
        d_P,
        pBuffer
    );
    assert( CUSPARSE_STATUS_SUCCESS == status);

/* step 5: gather sorted cooVals */
    status = cusparseDgthr(
        handle,
        nnz,
        d_cooVals,
        d_cooVals_sorted,
        d_P,
        CUSPARSE_INDEX_BASE_ZERO
    );
    assert( CUSPARSE_STATUS_SUCCESS == status);

    cudaStat1 = cudaDeviceSynchronize(); /* wait until the computation is done
 */
    cudaStat2 = cudaMemcpy(h_cooRows, d_cooRows, sizeof(int)*nnz    ,
 cudaMemcpyDeviceToHost);
    cudaStat3 = cudaMemcpy(h_cooCols, d_cooCols, sizeof(int)*nnz    ,
 cudaMemcpyDeviceToHost);
    cudaStat4 = cudaMemcpy(h_P,       d_P      , sizeof(int)*nnz    ,
 cudaMemcpyDeviceToHost);
    cudaStat5 = cudaMemcpy(h_cooVals, d_cooVals_sorted, sizeof(double)*nnz,
 cudaMemcpyDeviceToHost);
    cudaStat6 = cudaDeviceSynchronize();
    assert( cudaSuccess == cudaStat1 );
    assert( cudaSuccess == cudaStat2 );
    assert( cudaSuccess == cudaStat3 );
    assert( cudaSuccess == cudaStat4 );
    assert( cudaSuccess == cudaStat5 );
    assert( cudaSuccess == cudaStat6 );

    printf("sorted coo: \n");
    for(int j = 0 ; j < nnz; j++){
        printf("(%d, %d, %f) \n", h_cooRows[j], h_cooCols[j], h_cooVals[j] );
    }

    for(int j = 0 ; j < nnz; j++){
        printf("P[%d] = %d \n", j, h_P[j] );
    }

/* free resources */
    if (d_cooRows     ) cudaFree(d_cooRows);
    if (d_cooCols     ) cudaFree(d_cooCols);
    if (d_P           ) cudaFree(d_P);
    if (d_cooVals     ) cudaFree(d_cooVals);
    if (d_cooVals_sorted ) cudaFree(d_cooVals_sorted);
    if (pBuffer       ) cudaFree(pBuffer);
    if (handle        ) cusparseDestroy(handle);
    if (stream        ) cudaStreamDestroy(stream);
    cudaDeviceReset();
    return 0;
}
```

Chapter 18. APPENDIX E: EXAMPLES OF PRUNE

18.1. prune dense to sparse

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix of CSR format.

A is a 4x4 dense matrix,

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

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunedense_example.cpp
 *   g++ -o prunedense_example.cpp prunedense_example.o -L/usr/local/cuda/lib64
 -lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrC = NULL;
```

...

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int lda = m;
/*
 *       |    1     0     2     -3  |
 *       |    0     4     0     0   |
 *   A = |    5     0     6     7   |
 *       |    0     8     0     9   |
 *
 */
    const float A[lda*n] = {1, 0, 5, 0, 0, 4, 0, 8, 2, 0, 6, 0, -3, 0, 7, 9};
    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;

    float *d_A = NULL;
    int *d_csrRowPtrC = NULL;
    int *d_csrColIndC = NULL;
    float *d_csrValC = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    int nnzC = 0;

    float threshold = 4.1; /* remove Aij <= 4.1 */
//    float threshold = 0; /* remove zeros */

    printf("example of pruneDense2csr \n");

    printf("prune |A(i,j)| <= threshold \n");
    printf("threshold = %E \n", threshold);

    printMatrix(m, n, A, lda, "A");

/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);
```

...

```
/* step 2: configuration of matrix C */
    status = cusparseCreateMatDescr(&descrC);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    cusparseSetMatIndexBase(descrC,CUSPARSE_INDEX_BASE_ZERO);
    cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL );

    cudaStat1 = cudaMalloc ((void**)&d_A         , sizeof(float)*lda*n );
    cudaStat2 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);

/* step 3: query workspace */
    cudaStat1 = cudaMemcpy(d_A, A, sizeof(float)*lda*n, cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

    status = cusparseSpruneDense2csr_bufferSizeExt(
        handle,
        m,
        n,
        d_A,
        lda,
        &threshold,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);

    if (NULL != d_work) { cudaFree(d_work); }
    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
    status = cusparseSpruneDense2csrNnz(
        handle,
        m,
        n,
        d_A,
        lda,
        &threshold,
        descrC,
        d_csrRowPtrC,
        &nnzC, /* host */
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

    printf("nnzC = %d\n", nnzC);
    if (0 == nnzC ){
        printf("C is empty \n");
        return 0;
    }
```

...

```
/* step 5: compute csrColIndC and csrValC */
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
    cudaStat2 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);

    status = cusparseSpruneDense2csr(
        handle,
        m,
        n,
        d_A,
        lda,
        &threshold,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

/* step 6: output C */
    csrRowPtrC = (int*  )malloc(sizeof(int  )*(m+1));
    csrColIndC = (int*  )malloc(sizeof(int  )*nnzC);
    csrValC    = (float*)malloc(sizeof(float)*nnzC);
    assert( NULL != csrRowPtrC);
    assert( NULL != csrColIndC);
    assert( NULL != csrValC);

    cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
 cudaMemcpyDeviceToHost);
    cudaStat2 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
 cudaMemcpyDeviceToHost);
    cudaStat3 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);

    printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
    if (d_A           ) cudaFree(d_A);
    if (d_csrRowPtrC  ) cudaFree(d_csrRowPtrC);
    if (d_csrColIndC  ) cudaFree(d_csrColIndC);
    if (d_csrValC     ) cudaFree(d_csrValC);

    if (csrRowPtrC  ) free(csrRowPtrC);
    if (csrColIndC  ) free(csrColIndC);
    if (csrValC     ) free(csrValC);

    if (handle        ) cusparseDestroy(handle);
    if (stream        ) cudaStreamDestroy(stream);
    if (descrC        ) cusparseDestroyMatDescr(descrC);

    cudaDeviceReset();
    return 0;
}
```

18.2. prune sparse to sparse

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix of CSR format.

A is a 4x4 sparse matrix,

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

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunecsr_example.cpp
 *   g++ -o prunecsr_example.cpp prunecsr_example.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int nnzA = 9;
/*
 *       |    1     0     2     -3  |
 *       |    0     4     0     0   |
 *   A = |    5     0     6     7   |
 *       |    0     8     0     9   |
 *
 */
```

...

```
    const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
    const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
    const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;

    int *d_csrRowPtrA = NULL;
    int *d_csrColIndA = NULL;
    float *d_csrValA = NULL;

    int *d_csrRowPtrC = NULL;
    int *d_csrColIndC = NULL;
    float *d_csrValC = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    int nnzC = 0;

    float threshold = 4.1; /* remove Aij <= 4.1 */
//    float threshold = 0; /* remove zeros */

    printf("example of pruneCsr2csr \n");

    printf("prune |A(i,j)| <= threshold \n");
    printf("threshold = %E \n", threshold);

/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A and C */
    status = cusparseCreateMatDescr(&descrA);
    assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
    cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE);
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );

    status = cusparseCreateMatDescr(&descrC);
    assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
    cusparseSetMatIndexBase(descrC,CUSPARSE_INDEX_BASE_ZERO);
    cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL );

    printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");
```

...

```
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(float)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);

    cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(m+1),
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(float)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
    status = cusparseSpruneCsr2csr_bufferSizeExt(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        &threshold,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);

    if (NULL != d_work) { cudaFree(d_work); }
    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
    status = cusparseSpruneCsr2csrNnz(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        &threshold,
        descrC,
        d_csrRowPtrC,
        &nnzC, /* host */
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);
```

...

```
    printf("nnzC = %d\n", nnzC);
    if (0 == nnzC ){
        printf("C is empty \n");
        return 0;
    }
/* step 5: compute csrColIndC and csrValC */
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
    assert(cudaSuccess == cudaStat1);

    status = cusparseSpruneCsr2csr(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        &threshold,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

/* step 6: output C */
    csrRowPtrC = (int*  )malloc(sizeof(int  )*(m+1));
    csrColIndC = (int*  )malloc(sizeof(int  )*nnzC);
    csrValC    = (float*)malloc(sizeof(float)*nnzC);
    assert( NULL != csrRowPtrC);
    assert( NULL != csrColIndC);
    assert( NULL != csrValC);
    cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");
/* free resources */
    if (d_csrRowPtrA  ) cudaFree(d_csrRowPtrA);
    if (d_csrColIndA  ) cudaFree(d_csrColIndA);
    if (d_csrValA     ) cudaFree(d_csrValA);
    if (d_csrRowPtrC  ) cudaFree(d_csrRowPtrC);
    if (d_csrColIndC  ) cudaFree(d_csrColIndC);
    if (d_csrValC     ) cudaFree(d_csrValC);
    if (csrRowPtrC  ) free(csrRowPtrC);
    if (csrColIndC  ) free(csrColIndC);
    if (csrValC     ) free(csrValC);
    if (handle        ) cusparseDestroy(handle);
    if (stream        ) cudaStreamDestroy(stream);
    if (descrA        ) cusparseDestroyMatDescr(descrA);
    if (descrC        ) cusparseDestroyMatDescr(descrC);
    cudaDeviceReset();
    return 0;
}
```

18.3. prune dense to sparse by percentage

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix by percentage.

A is a 4x4 dense matrix,

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

The percentage is 50, which means to prune 50 percent of the dense matrix. The matrix has 16 elements, so 8 out of 16 must be pruned out. Therefore 7 zeros are pruned out, and value 1.0 is also out because it is the smallest among 9 nonzero elements.

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunedense2csrbyP.cpp
 *   g++ -o prunedense2csrbyP.cpp prunedense2csrbyP.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrC = NULL;
    pruneInfo_t info = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int lda = m;
```

...

```
/*
 *       |    1     0     2     -3  |
 *       |    0     4     0     0   |
 *  A =  |    5     0     6     7   |
 *       |    0     8     0     9   |
 *
 */
    const float A[lda*n] = {1, 0, 5, 0, 0, 4, 0, 8, 2, 0, 6, 0, -3, 0, 7, 9};
    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;

    float *d_A = NULL;
    int *d_csrRowPtrC = NULL;
    int *d_csrColIndC = NULL;
    float *d_csrValC = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    int nnzC = 0;

    float percentage = 50; /* 50% of nnz */

    printf("example of pruneDense2csrByPercentage \n");

    printf("prune out %.1f percentage of A \n", percentage);

    printMatrix(m, n, A, lda, "A");

/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseCreatePruneInfo(&info);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix C */
    status = cusparseCreateMatDescr(&descrC);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    cusparseSetMatIndexBase(descrC,CUSPARSE_INDEX_BASE_ZERO);
    cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL );

    cudaStat1 = cudaMalloc ((void**)&d_A         , sizeof(float)*lda*n );
    cudaStat2 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);

    cudaStat1 = cudaMemcpy(d_A, A, sizeof(float)*lda*n, cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
```

...

```
/* step 3: query workspace */
    status = cusparseSpruneDense2csrByPercentage_bufferSizeExt(
        handle,
        m,
        n,
        d_A,
        lda,
        percentage,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        info,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes = %lld \n", (long long)lworkInBytes);

    if (NULL != d_work) { cudaFree(d_work); }
    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
    status = cusparseSpruneDense2csrNnzByPercentage(
        handle,
        m,
        n,
        d_A,
        lda,
        percentage,
        descrC,
        d_csrRowPtrC,
        &nnzC, /* host */
        info,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

    printf("nnzC = %d\n", nnzC);
    if (0 == nnzC ){
        printf("C is empty \n");
        return 0;
    }

/* step 5: compute csrColIndC and csrValC */
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
    cudaStat2 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
```

...

```
    status = cusparseSpruneDense2csrByPercentage(
        handle,
        m,
        n,
        d_A,
        lda,
        percentage,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        info,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

/* step 7: output C */
    csrRowPtrC = (int*  )malloc(sizeof(int  )*(m+1));
    csrColIndC = (int*  )malloc(sizeof(int  )*nnzC);
    csrValC    = (float*)malloc(sizeof(float)*nnzC);
    assert( NULL != csrRowPtrC);
    assert( NULL != csrColIndC);
    assert( NULL != csrValC);

    cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
 cudaMemcpyDeviceToHost);
    cudaStat2 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
 cudaMemcpyDeviceToHost);
    cudaStat3 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    assert(cudaSuccess == cudaStat2);
    assert(cudaSuccess == cudaStat3);

    printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
    if (d_A         ) cudaFree(d_A);
    if (d_csrRowPtrC) cudaFree(d_csrRowPtrC);
    if (d_csrColIndC) cudaFree(d_csrColIndC);
    if (d_csrValC   ) cudaFree(d_csrValC);

    if (csrRowPtrC  ) free(csrRowPtrC);
    if (csrColIndC  ) free(csrColIndC);
    if (csrValC     ) free(csrValC);

    if (handle      ) cusparseDestroy(handle);
    if (stream      ) cudaStreamDestroy(stream);
    if (descrC      ) cusparseDestroyMatDescr(descrC);
    if (info        ) cusparseDestroyPruneInfo(info);

    cudaDeviceReset();

    return 0;
}
```

18.4. prune sparse to sparse by percentage

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix by percentage.

A is a 4x4 sparse matrix,

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

The percentage is 20, which means to prune 20 percent of the nonzeros. The sparse matrix has 9 nonzero elements, so 1.4 elements must be pruned out. The function removes 1.0 and 2.0 which are first two smallest numbers of nonzeros.

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunecsr2csrByP.cpp
 *   g++ -o prunecsr2csrByP.cpp prunecsr2csrByP.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;
    pruneInfo_t info = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int nnzA = 9;
/*
 *       |    1     0     2     -3  |
 *       |    0     4     0     0   |
 *  A =  |    5     0     6     7   |
 *       |    0     8     0     9   |
 *
 */
```

...

```
    const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
    const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
    const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;

    int *d_csrRowPtrA = NULL;
    int *d_csrColIndA = NULL;
    float *d_csrValA = NULL;

    int *d_csrRowPtrC = NULL;
    int *d_csrColIndC = NULL;
    float *d_csrValC = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    int nnzC = 0;

    float percentage = 20; /* remove 20% of nonzeros */

    printf("example of pruneCsr2csrByPercentage \n");

    printf("prune %.1f percent of nonzeros \n", percentage);

/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseCreatePruneInfo(&info);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix C */
    status = cusparseCreateMatDescr(&descrA);
    assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
    cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE);
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );

    status = cusparseCreateMatDescr(&descrC);
    assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
    cusparseSetMatIndexBase(descrC,CUSPARSE_INDEX_BASE_ZERO);
    cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL );

    printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");
```

...

```
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(float)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
    assert(cudaSuccess == cudaStat1);

    cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(m+1),
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(float)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
    status = cusparseSpruneCsr2csrByPercentage_bufferSizeExt(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        percentage,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        info,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes = %lld \n", (long long)lworkInBytes);

    if (NULL != d_work) { cudaFree(d_work); }
    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
    status = cusparseSpruneCsr2csrNnzByPercentage(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        percentage,
        descrC,
        d_csrRowPtrC,
        &nnzC, /* host */
        info,
        d_work);
```

...

```
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

    printf("nnzC = %d\n", nnzC);
    if (0 == nnzC ){
        printf("C is empty \n");
        return 0;
    }

/* step 5: compute csrColIndC and csrValC */
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
    assert(cudaSuccess == cudaStat1);

    status = cusparseSpruneCsr2csrByPercentage(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        percentage,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        info,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

/* step 6: output C */
    csrRowPtrC = (int*  )malloc(sizeof(int  )*(m+1));
    csrColIndC = (int*  )malloc(sizeof(int  )*nnzC);
    csrValC    = (float*)malloc(sizeof(float)*nnzC);
    assert( NULL != csrRowPtrC);
    assert( NULL != csrColIndC);
    assert( NULL != csrValC);

    cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);

    printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");
```

...

```
/* free resources */
    if (d_csrRowPtrA) cudaFree(d_csrRowPtrA);
    if (d_csrColIndA) cudaFree(d_csrColIndA);
    if (d_csrValA   ) cudaFree(d_csrValA);
    if (d_csrRowPtrC) cudaFree(d_csrRowPtrC);
    if (d_csrColIndC) cudaFree(d_csrColIndC);
    if (d_csrValC   ) cudaFree(d_csrValC);

    if (csrRowPtrC  ) free(csrRowPtrC);
    if (csrColIndC  ) free(csrColIndC);
    if (csrValC     ) free(csrValC);

    if (handle      ) cusparseDestroy(handle);
    if (stream      ) cudaStreamDestroy(stream);
    if (descrA      ) cusparseDestroyMatDescr(descrA);
    if (descrC      ) cusparseDestroyMatDescr(descrC);
    if (info        ) cusparseDestroyPruneInfo(info);

    cudaDeviceReset();

    return 0;
}
```

Chapter 19.
APPENDIX F: EXAMPLES OF GTSV

19.1. batched tridiagonal solver

This section provides a simple example in the C programming language of gtsvInterleavedBatch .

The example solves two linear systems and assumes data layout is NOT interleaved format. Before calling gtsvInterleavedBatch, cublasXgeam is used to transform the data layout, from aggregate format to interleaved format. If the user can prepare interleaved format, no need to transpose the data.

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include gtsv.cpp
 *   g++ -o gtsv gtsv.o -L/usr/local/cuda/lib64 -lcusparse -lcublas -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>
#include <cublas_v2.h>

/*
 * compute | b - A*x|_inf
 */
void residaul_eval(
    int n,
    const float *dl,
    const float *d,
    const float *du,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    float r_nrminf = 0;
    for(int i = 0 ; i < n ; i++){
        float dot = 0;
        if (i > 0 ){
            dot += dl[i]*x[i-1];
        }
        dot += d[i]*x[i];
        if (i < (n-1) ){
            dot += du[i]*x[i+1];
        }
        float ri = b[i] - dot;
        r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    }

    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t cusparseH = NULL;
    cublasHandle_t cublasH = NULL;
    cudaStream_t stream = NULL;
```

...

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;

    const int n = 3;
    const int batchSize = 2;
/*
 *       |    1     6     0  |        | 1 |        | -0.603960 |
 *  A1 =|    4     2     7  |, b1 = | 2 |, x1 = |  0.267327 |
 *       |    0     5     3  |        | 3 |        |  0.554455 |
 *
 *       |    8    13     0  |        | 4 |        | -0.063291 |
 *  A2 =|   11     9    14  |, b2 = | 5 |, x2 = |  0.346641 |
 *       |    0    12    10  |        | 6 |        |  0.184031 |
 */

/*
 * A = (dl, d, du), B and X are in aggregate format
 */
    const float dl[n * batchSize] = {0, 4, 5,  0, 11, 12};
    const float  d[n * batchSize] = {1, 2, 3,  8,  9, 10};
    const float du[n * batchSize] = {6, 7, 0, 13, 14,  0};
    const float  B[n * batchSize] = {1, 2, 3,  4,  5,  6};
    float X[n * batchSize]; /* Xj = Aj \ Bj */

/* device memory
 * (d_dl0, d_d0, d_du0) is aggregate format
 * (d_dl, d_d, d_du) is interleaved format
 */
    float *d_dl0 = NULL;
    float *d_d0  = NULL;
    float *d_du0 = NULL;
    float *d_dl  = NULL;
    float *d_d   = NULL;
    float *d_du  = NULL;
    float *d_B   = NULL;
    float *d_X   = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

/*
 * algo = 0: cuThomas (unstable)
 * algo = 1: LU with pivoting (stable)
 * algo = 2: QR (stable)
 */
    const int algo = 2;

    const float h_one = 1;
    const float h_zero = 0;

    printf("example of gtsv (interleaved format) \n");
    printf("choose algo = 0,1,2 to select different algorithms \n");
    printf("n = %d, batchSize = %d, algo = %d \n", n, batchSize, algo);
```

...

```
/* step 1: create cusparse/cublas handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);
    status = cusparseCreate(&cusparseH);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    status = cusparseSetStream(cusparseH, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cublasStat = cublasCreate(&cublasH);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
    cublasStat = cublasSetStream(cublasH, stream);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* step 2: allocate device memory */
    cudaStat1 = cudaMalloc ((void**)&d_dl0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_d0  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_du0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_dl  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_d   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_du  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_B   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_X   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);

/* step 3: prepare data in device, interleaved format */
    cudaStat1 = cudaMemcpy(d_dl0, dl, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_d0 , d , sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_du0, du, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_B  ,  B, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

    cudaDeviceSynchronize();
    /* convert dl to interleaved format
     *  dl = transpose(dl0)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of dl */
        n,         /* number of columns of dl */
        &h_one,
        d_dl0,  /* dl0 is n-by-batchSize */
        n, /* leading dimension of dl0 */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_dl,      /* dl is batchSize-by-n */
        batchSize  /* leading dimension of dl */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
```

...

```
    /* convert d to interleaved format
     *  d = transpose(d0)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of d */
        n,         /* number of columns of d */
        &h_one,
        d_d0, /* d0 is n-by-batchSize */
        n, /* leading dimension of d0 */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_d,       /* d is batchSize-by-n */
        batchSize  /* leading dimension of d */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    /* convert du to interleaved format
     *  du = transpose(du0)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of du */
        n,         /* number of columns of du */
        &h_one,
        d_du0, /* du0 is n-by-batchSize */
        n, /* leading dimension of du0 */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_du,      /* du is batchSize-by-n */
        batchSize  /* leading dimension of du */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    /* convert B to interleaved format
     *  X = transpose(B)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of X */
        n,         /* number of columns of X */
        &h_one,
        d_B, /* B is n-by-batchSize */
        n, /* leading dimension of B */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_X,       /* X is batchSize-by-n */
        batchSize  /* leading dimension of X */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
```

...

```
/* step 4: prepare workspace */
    status = cusparseSgtsvInterleavedBatch_bufferSizeExt(
        cusparseH,
        algo,
        n,
        d_dl,
        d_d,
        d_du,
        d_X,
        batchSize,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes = %lld \n", (long long)lworkInBytes);

    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 5: solve Aj*xj = bj */
    status = cusparseSgtsvInterleavedBatch(
        cusparseH,
        algo,
        n,
        d_dl,
        d_d,
        d_du,
        d_X,
        batchSize,
        d_work);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSPARSE_STATUS_SUCCESS == status);
    assert(cudaSuccess == cudaStat1);

/* step 6: convert X back to aggregate format  */
    /* B = transpose(X) */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        n,         /* number of rows of B */
        batchSize, /* number of columns of B */
        &h_one,
        d_X,       /* X is batchSize-by-n */
        batchSize, /* leading dimension of X */
        &h_zero,
        NULL,
        n, /* don't cae */
        d_B, /* B is n-by-batchSize */
        n  /* leading dimension of B */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    cudaDeviceSynchronize();

/* step 7: residual evaluation */
    cudaStat1 = cudaMemcpy(X,  d_B, sizeof(float)*n*batchSize,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);

    cudaDeviceSynchronize();
```

...

```
    printf("==== x1 = inv(A1)*b1 \n");
    for(int j = 0 ; j < n; j++){
        printf("x1[%d] = %f\n", j, X[j]);
    }

    float r1_nrminf;
    residaul_eval(
        n,
        dl,
        d,
        du,
        B,
        X,
        &r1_nrminf
    );
    printf("|b1 - A1*x1| = %E\n", r1_nrminf);

    printf("\n==== x2 = inv(A2)*b2 \n");
    for(int j = 0 ; j < n; j++){
        printf("x2[%d] = %f\n", j, X[n+j]);
    }

    float r2_nrminf;
    residaul_eval(
        n,
        dl + n,
        d + n,
        du + n,
        B + n,
        X + n,
        &r2_nrminf
    );
    printf("|b2 - A2*x2| = %E\n", r2_nrminf);

/* free resources */
    if (d_dl0       ) cudaFree(d_dl0);
    if (d_d0        ) cudaFree(d_d0);
    if (d_du0       ) cudaFree(d_du0);
    if (d_dl        ) cudaFree(d_dl);
    if (d_d         ) cudaFree(d_d);
    if (d_du        ) cudaFree(d_du);
    if (d_B         ) cudaFree(d_B);
    if (d_X         ) cudaFree(d_X);

    if (cusparseH   ) cusparseDestroy(cusparseH);
    if (cublasH     ) cublasDestroy(cublasH);
    if (stream      ) cudaStreamDestroy(stream);

    cudaDeviceReset();

    return 0;
}
```

Chapter 20. APPENDIX G: EXAMPLES OF GPSV

20.1. batched penta-diagonal solver

This section provides a simple example in the C programming language of gpsvInterleavedBatch .

The example solves two penta-diagonal systems and assumes data layout is NOT interleaved format. Before calling gpsvInterleavedBatch, cublasXgeam is used to transform the data layout, from aggregate format to interleaved format. If the user can prepare interleaved format, no need to transpose the data.

...

```
*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include gpsv.cpp
 *   g++ -o gpsv gpsv.o -L/usr/local/cuda/lib64 -lcusparse -lcublas -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>
#include <cublas_v2.h>

/*
 * compute | b - A*x|_inf
 */
void residaul_eval(
    int n,
    const float *ds,
    const float *dl,
    const float *d,
    const float *du,
    const float *dw,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    float r_nrminf = 0;
    for(int i = 0 ; i < n ; i++){
        float dot = 0;
        if (i > 1 ){
            dot += ds[i]*x[i-2];
        }
        if (i > 0 ){
            dot += dl[i]*x[i-1];
        }
        dot += d[i]*x[i];
        if (i < (n-1) ){
            dot += du[i]*x[i+1];
        }
        if (i < (n-2) ){
            dot += dw[i]*x[i+2];
        }
        float ri = b[i] - dot;
        r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    }

    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t cusparseH = NULL;
    cublasHandle_t cublasH = NULL;
    cudaStream_t stream = NULL;
```

...

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;

    const int n = 4;
    const int batchSize = 2;

/*
 *       |  1    8   13   0  |        | 1 |        | -0.0592 |
 *  A1 =|  5    2    9  14  |, b1 = | 2 |, x1 = |  0.3428 |
 *       | 11    6    3  10  |        | 3 |        | -0.1295 |
 *       |  0   12    7   4  |        | 4 |        |  0.1982 |
 *
 *       | 15   22   27   0  |        | 5 |        | -0.0012 |
 *  A2 =| 19   16   23  28  |, b2 = | 6 |, x2 = |  0.2792 |
 *       | 25   20   17  24  |        | 7 |        | -0.0416 |
 *       |  0   26   21  18  |        | 8 |        |  0.0898 |
 */

/*
 * A = (ds, dl, d, du, dw), B and X are in aggregate format
 */
    const float ds[n * batchSize] = { 0, 0, 11, 12,  0,  0, 25, 26};
    const float dl[n * batchSize] = { 0, 5,  6,  7,  0, 19, 20, 21};
    const float  d[n * batchSize] = { 1, 2,  3,  4, 15, 16, 17, 18};
    const float du[n * batchSize] = { 8, 9, 10,  0, 22, 23, 24,  0};
    const float dw[n * batchSize] = {13,14,  0,  0, 27, 28,  0,  0};
    const float  B[n * batchSize] = { 1, 2,  3,  4,  5,  6,  7,  8};
    float X[n * batchSize]; /* Xj = Aj \ Bj */

/* device memory
 * (d_ds0, d_dl0, d_d0, d_du0, d_dw0) is aggregate format
 * (d_ds, d_dl, d_d, d_du, d_dw) is interleaved format
 */
    float *d_ds0 = NULL;
    float *d_dl0 = NULL;
    float *d_d0  = NULL;
    float *d_du0 = NULL;
    float *d_dw0 = NULL;
    float *d_ds  = NULL;
    float *d_dl  = NULL;
    float *d_d   = NULL;
    float *d_du  = NULL;
    float *d_dw  = NULL;
    float *d_B   = NULL;
    float *d_X   = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    const float h_one = 1;
    const float h_zero = 0;

    int algo = 0 ; /* QR factorization */

    printf("example of gpsv (interleaved format) \n");
    printf("n = %d, batchSize = %d\n", n, batchSize);

/* step 1: create cusparse/cublas handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&cusparseH);
    assert(CUSPARSE_STATUS_SUCCESS == status);
```

...

```
    status = cusparseSetStream(cusparseH, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cublasStat = cublasCreate(&cublasH);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
    cublasStat = cublasSetStream(cublasH, stream);
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* step 2: allocate device memory */
    cudaStat1 = cudaMalloc ((void**)&d_ds0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_dl0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_d0  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_du0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_dw0 , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_ds  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_dl  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_d   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_du  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_dw  , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_B   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_X   , sizeof(float)*n*batchSize );
    assert(cudaSuccess == cudaStat1);
/* step 3: prepare data in device, interleaved format */
    cudaStat1 = cudaMemcpy(d_ds0, ds, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_dl0, dl, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_d0 , d , sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_du0, du, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_dw0, dw, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_B  ,  B, sizeof(float)*n*batchSize,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaDeviceSynchronize();
   /* convert ds to interleaved format
    *  ds = transpose(ds0) */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of ds */
        n,         /* number of columns of ds */
        &h_one,
        d_ds0,  /* ds0 is n-by-batchSize */
        n, /* leading dimension of ds0 */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_ds,      /* ds is batchSize-by-n */
        batchSize);  /* leading dimension of ds */
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
```

...

```
/* convert dl to interleaved format
 *  dl = transpose(dl0)
 */
 cublasStat = cublasSgeam(
     cublasH,
     CUBLAS_OP_T, /* transa */
     CUBLAS_OP_T, /* transb, don't care */
     batchSize, /* number of rows of dl */
     n,         /* number of columns of dl */
     &h_one,
     d_dl0,  /* dl0 is n-by-batchSize */
     n, /* leading dimension of dl0 */
     &h_zero,
     NULL,
     n,         /* don't cae */
     d_dl,      /* dl is batchSize-by-n */
     batchSize  /* leading dimension of dl */
 );
 assert(CUBLAS_STATUS_SUCCESS == cublasStat);

 /* convert d to interleaved format
  *  d = transpose(d0)
  */
 cublasStat = cublasSgeam(
     cublasH,
     CUBLAS_OP_T, /* transa */
     CUBLAS_OP_T, /* transb, don't care */
     batchSize, /* number of rows of d */
     n,         /* number of columns of d */
     &h_one,
     d_d0, /* d0 is n-by-batchSize */
     n, /* leading dimension of d0 */
     &h_zero,
     NULL,
     n,         /* don't cae */
     d_d,       /* d is batchSize-by-n */
     batchSize  /* leading dimension of d */
 );
 assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* convert du to interleaved format
  *  du = transpose(du0)
  */
 cublasStat = cublasSgeam(
     cublasH,
     CUBLAS_OP_T, /* transa */
     CUBLAS_OP_T, /* transb, don't care */
     batchSize, /* number of rows of du */
     n,         /* number of columns of du */
     &h_one,
     d_du0, /* du0 is n-by-batchSize */
     n, /* leading dimension of du0 */
     &h_zero,
     NULL,
     n,         /* don't cae */
     d_du,      /* du is batchSize-by-n */
     batchSize  /* leading dimension of du */
 );
 assert(CUBLAS_STATUS_SUCCESS == cublasStat);
```

...

```
   /* convert dw to interleaved format
     *  dw = transpose(dw0)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of dw */
        n,         /* number of columns of dw */
        &h_one,
        d_dw0, /* dw0 is n-by-batchSize */
        n, /* leading dimension of dw0 */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_dw,      /* dw is batchSize-by-n */
        batchSize  /* leading dimension of dw */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

    /* convert B to interleaved format
     *  X = transpose(B)
     */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        batchSize, /* number of rows of X */
        n,         /* number of columns of X */
        &h_one,
        d_B, /* B is n-by-batchSize */
        n, /* leading dimension of B */
        &h_zero,
        NULL,
        n,         /* don't cae */
        d_X,       /* X is batchSize-by-n */
        batchSize  /* leading dimension of X */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* step 4: prepare workspace */
    status = cusparseSgpsvInterleavedBatch_bufferSizeExt(
        cusparseH,
        algo,
        n,
        d_ds,
        d_dl,
        d_d,
        d_du,
        d_dw,
        d_X,
        batchSize,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes = %lld \n", (long long)lworkInBytes);

    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);
```

...

```
/* step 5: solve Aj*xj = bj */
    status = cusparseSgpsvInterleavedBatch(
        cusparseH,
        algo,
        n,
        d_ds,
        d_dl,
        d_d,
        d_du,
        d_dw,
        d_X,
        batchSize,
        d_work);
    cudaStat1 = cudaDeviceSynchronize();
    assert(CUSPARSE_STATUS_SUCCESS == status);
    assert(cudaSuccess == cudaStat1);

/* step 6: convert X back to aggregate format  */
    /* B = transpose(X) */
    cublasStat = cublasSgeam(
        cublasH,
        CUBLAS_OP_T, /* transa */
        CUBLAS_OP_T, /* transb, don't care */
        n,         /* number of rows of B */
        batchSize, /* number of columns of B */
        &h_one,
        d_X,       /* X is batchSize-by-n */
        batchSize, /* leading dimension of X */
        &h_zero,
        NULL,
        n, /* don't cae */
        d_B, /* B is n-by-batchSize */
        n  /* leading dimension of B */
    );
    assert(CUBLAS_STATUS_SUCCESS == cublasStat);
    cudaDeviceSynchronize();

/* step 7: residual evaluation */
    cudaStat1 = cudaMemcpy(X,  d_B, sizeof(float)*n*batchSize,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaDeviceSynchronize();

    printf("==== x1 = inv(A1)*b1 \n");
    for(int j = 0 ; j < n; j++){
        printf("x1[%d] = %f\n", j, X[j]);
    }

    float r1_nrminf;
    residaul_eval(
        n,
        ds,
        dl,
        d,
        du,
        dw,
        B,
        X,
        &r1_nrminf
    );
    printf("|b1 - A1*x1| = %E\n", r1_nrminf);
```

...

```
    printf("\n==== x2 = inv(A2)*b2 \n");
    for(int j = 0 ; j < n; j++){
        printf("x2[%d] = %f\n", j, X[n+j]);
    }

    float r2_nrminf;
    residaul_eval(
        n,
        ds + n,
        dl + n,
        d + n,
        du + n,
        dw + n,
        B + n,
        X + n,
        &r2_nrminf
    );
    printf("|b2 - A2*x2| = %E\n", r2_nrminf);

/* free resources */
    if (d_ds0       ) cudaFree(d_ds0);
    if (d_dl0       ) cudaFree(d_dl0);
    if (d_d0        ) cudaFree(d_d0);
    if (d_du0       ) cudaFree(d_du0);
    if (d_dw0       ) cudaFree(d_dw0);
    if (d_ds        ) cudaFree(d_ds);
    if (d_dl        ) cudaFree(d_dl);
    if (d_d         ) cudaFree(d_d);
    if (d_du        ) cudaFree(d_du);
    if (d_dw        ) cudaFree(d_dw);
    if (d_B         ) cudaFree(d_B);
    if (d_X         ) cudaFree(d_X);

    if (cusparseH   ) cusparseDestroy(cusparseH);
    if (cublasH     ) cublasDestroy(cublasH);
    if (stream      ) cudaStreamDestroy(stream);

    cudaDeviceReset();

    return 0;
}
```

Chapter 21.
APPENDIX H: EXAMPLES OF CSRSM2

21.1. forward triangular solver

This section provides a simple example in the C programming language of csrsm2.

The example solves a lower triangular system with 2 right hand side vectors.

...

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include csrms2.cpp
 *   g++ -o csrm2 csrsm2.o -L/usr/local/cuda/lib64 -lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

/* compute | b - A*x|_inf */
void residaul_eval(
    int n,
    const cusparseMatDescr_t descrA,
    const float *csrVal,
    const int *csrRowPtr,
    const int *csrColInd,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;
    const int lower = (CUSPARSE_FILL_MODE_LOWER ==
 cusparseGetMatFillMode(descrA))? 1:0;
    const int unit  = (CUSPARSE_DIAG_TYPE_UNIT ==
 cusparseGetMatDiagType(descrA))? 1:0;

    float r_nrminf = 0;
    for(int row = 0 ; row < n ; row++){
        const int start = csrRowPtr[row]   - base;
        const int end   = csrRowPtr[row+1] - base;
        float dot = 0;
        for(int colidx = start ; colidx < end; colidx++){
            const int col = csrColInd[colidx] - base;
            float Aij = csrVal[colidx];
            float xj  = x[col];
            if ( (row == col) && unit ){
                Aij = 1.0;
            }
            int valid = (row >= col) && lower ||
                        (row <= col) && !lower ;
            if ( valid ){
                dot += Aij*xj;
            }
        }
        float ri = b[row] - dot;
        r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    }
    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    csrsm2Info_t info = NULL;
```

...

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int nrhs = 2;
    const int n = 4;
    const int nnzA = 9;
    const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
    const float h_one = 1.0;
/*
 *      |    1     0     2     -3  |
 *      |    0     4     0     0   |
 *  A = |    5     0     6     7   |
 *      |    0     8     0     9   |
 *
 *  Regard A as a lower triangle matrix L with non-unit diagonal.
 *             | 1  5 |              | 1          5      |
 *  Given  B = | 2  6 |, X = L \ B = | 0.5       1.5     |
 *             | 3  7 |              | -0.3333   -3      |
 *             | 4  8 |              |  0        -0.4444 |
 */
    const int csrRowPtrA[n+1] = { 1, 4, 5, 8, 10};
    const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
    const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};
    const float B[n*nrhs] = {1,2,3,4,5,6,7,8};
    float X[n*nrhs];

    int *d_csrRowPtrA = NULL;
    int *d_csrColIndA = NULL;
    float *d_csrValA = NULL;
    float *d_B = NULL;

    size_t lworkInBytes = 0;
    char *d_work = NULL;

    const int algo = 0; /* non-block version */

    printf("example of csrsm2 \n");

/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseCreateCsrsm2Info(&info);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A */
    status = cusparseCreateMatDescr(&descrA);
    assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
    cusparseSetMatIndexBase(descrA,CUSPARSE_INDEX_BASE_ONE);

    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
/* A is lower triangle */
    cusparseSetMatFillMode(descrA, CUSPARSE_FILL_MODE_LOWER);
/* A has non unit diagonal */
    cusparseSetMatDiagType(descrA, CUSPARSE_DIAG_TYPE_NON_UNIT);
```

...

```
    cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(n+1) );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(float)*nnzA );
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_B         , sizeof(float)*n*nrhs );
    assert(cudaSuccess == cudaStat1);

    cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(n+1),
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(float)*nnzA,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(d_B         , B         , sizeof(float)*n*nrhs,
 cudaMemcpyHostToDevice);
    assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
    status = cusparseScsrsm2_bufferSizeExt(
        handle,
        algo,
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
        n,
        nrhs,
        nnzA,
        &h_one,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        d_B,
        n,   /* ldb */
        info,
        policy,
        &lworkInBytes);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    printf("lworkInBytes  = %lld \n", (long long)lworkInBytes);
    if (NULL != d_work) { cudaFree(d_work); }
    cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
    assert(cudaSuccess == cudaStat1);

/* step 4: analysis */
    status = cusparseScsrsm2_analysis(
        handle,
        algo,
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
        n,
        nrhs,
        nnzA,
        &h_one,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        d_B,
        n,   /* ldb */
        info,
        policy,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
```

...

```
/* step 5: solve L * X = B */
    status = cusparseScsrsm2_solve(
        handle,
        algo,
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
        CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
        n,
        nrhs,
        nnzA,
        &h_one,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        d_B,
        n,   /* ldb */
        info,
        policy,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

/* step 6:measure residual B - A*X */
    cudaStat1 = cudaMemcpy(X, d_B, sizeof(float)*n*nrhs,
 cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaDeviceSynchronize();

    printf("==== x1 = inv(A)*b1 \n");
    for(int j = 0 ; j < n; j++){
        printf("x1[%d] = %f\n", j, X[j]);
    }
    float r1_nrminf;
    residaul_eval(
        n,
        descrA,
        csrValA,
        csrRowPtrA,
        csrColIndA,
        B,
        X,
        &r1_nrminf
    );
    printf("|b1 - A*x1| = %E\n", r1_nrminf);

    printf("==== x2 = inv(A)*b2 \n");
    for(int j = 0 ; j < n; j++){
        printf("x2[%d] = %f\n", j, X[n+j]);
    }
    float r2_nrminf;
    residaul_eval(
        n,
        descrA,
        csrValA,
        csrRowPtrA,
        csrColIndA,
        B+n,
        X+n,
        &r2_nrminf
    );
    printf("|b2 - A*x2| = %E\n", r2_nrminf);
```

...

```
/* free resources */
    if (d_csrRowPtrA  ) cudaFree(d_csrRowPtrA);
    if (d_csrColIndA  ) cudaFree(d_csrColIndA);
    if (d_csrValA     ) cudaFree(d_csrValA);
    if (d_B           ) cudaFree(d_B);

    if (handle        ) cusparseDestroy(handle);
    if (stream        ) cudaStreamDestroy(stream);
    if (descrA        ) cusparseDestroyMatDescr(descrA);
    if (info          ) cusparseDestroyCsrsm2Info(info);

    cudaDeviceReset();

    return 0;
}
```

Chapter 22. APPENDIX I: ACKNOWLEDGEMENTS

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

- The cusparse<t>gtsv implementation is derived from a version developed by Li-Wen Chang from the University of Illinois.
- the cusparse<t>gtsvInterleavedBatch adopts cuThomasBatch developed by Pedro Valero-Lara and Ivan Martínez-Pérez from Barcelona Supercomputing Center and BSC/UPC NVIDIA GPU Center of Excellence.

Chapter 23. BIBLIOGRAPHY

[1] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors", Supercomputing, 2009.

[2] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 User's Guide", Technical Report CNA-150, Center for Numerical Analysis, University of Texas, 1979.

[3] M. Naumov, "Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS", Technical Report and White Paper, 2011.

[4] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J. Peña. NVIDIA GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems. Implementation of cuThomasBatch. In Parallel Processing and Applied Mathematics - 12th International Conference (PPAM), 2017.