CUDA Toolkit 5.0
CUSPARSE Library

PG-05329-050_v01 | June 2012

Contents

Published by
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Notice

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

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

Trademarks

NVIDIA, CUDA, and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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 (that is part of 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 (that can also usually be viewed as a dense tall matrix).
- Conversion: operations that allow conversion between different matrix formats.

The CUSPARSE library allows the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU), but does not auto-parallelize across multiple GPUs. The CUSPARSE API assumes that the input and output data reside in GPU (device) memory, unless specifically indicated otherwise by the string `DevHostPtr` being part of the parameter name of a function (for example, `*resultDevHostPtr` in `cusparse<t>doti`).

It is the responsibility of the user 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 1.1 or higher. Please see *NVIDIA CUDA C Programming Guide*, Appendix A for the list of the compute capabilities corresponding to all NVIDIA GPUs.

Note: The CUSPARSE library requires hardware with CC of at least 1.1.

1.1 New and Legacy CUSPARSE API

Starting with version 4.1, the CUSPARSE Library provides a new updated API, in addition to the existing legacy API. This section discusses why a new API is provided, the advantages of using it, and the differences with the existing legacy API.

The new CUSPARSE library API can be used by including the header file "cusparse_v2.h". It has the following features that the legacy CUSPARSE API does not have:

- the scalars α and β can be passed by reference on the host or the device, instead of only being allowed to be passed by value on the host. This change allows library functions to execute asynchronously using streams even when α and β are generated by a previous kernel.
- when a library routine returns a scalar result, it can be returned by reference on the host or the device, instead of only being allowed to be returned by value only on the host. This change allows library routines to be called asynchronously when the scalar result is generated and returned by reference on the device resulting in maximum parallelism.
- the function `cusparseSetKernelStream()` was renamed `cusparseSetStream()` to be more consistent with the other CUDA libraries.
- the enum type `cusparseAction_t` was introduced to indicate if the routine operates only on indices or on values and indices at the same time.

The legacy CUSPARSE API, explained in more detail in the Appendix A, can be used by including the header file "cusparse.h". Since the legacy API is identical to the previously released CUSPARSE library API, existing applications will work out of the box and automatically use this legacy API without any source code changes. In general, new applications should not use the legacy CUSPARSE API, and existing applications should convert to using the new API if it requires sophisticated and optimal stream parallelism. For the rest of the document, the new CUSPARSE Library API will simply be referred to as the CUSPARSE Library API.

As mentioned earlier the interfaces to the legacy and the CUSPARSE library APIs are the header file "cusparse.h" and "cusparse_v2.h", respectively. In addition, applications using the CUSPARSE library need to link against the dynamic shared object (DSO) cusparse.so on Linux, the dynamic-link library (DLL) cusparse.dll on Windows, or the dynamic library cusparse.dylib on Mac OS X. Note that the same dynamic library implements both the new and legacy CUSPARSE APIs.

1.2 Naming Convention

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 the 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 generic type, repectively.

The `<matrix data format>` can be `dense`, `coo`, `csr`, `csc` and `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` and `sctr`, corresponding to the Level 1 functions. Also, it can be `mv` and `sv` as well as `mm` and `sm`, corresponding pair-wise to the Level 2 and 3 functions, respectively.

All of these functions have the return type `cusparseStatus_t` and will be explained in more detail in the following chapters.

1.3 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. The user can use `cudaDeviceSynchronize` function to ensure that the execution of a particular CUSPARSE library routine has completed.

The application can also use `cudaMemcpy` routine to copy data from the device to host and vice-versa, using `cudaMemcpyDeviceToHost` and `cudaMemcpyHostToDevice` parameters, respectively. In this case there is no need to add a call to `cudaDeviceSynchronize`, because the call to `cudaMemcpy` routine with the above parameters is blocking and will only complete when the results are ready on the host.

Chapter 2
Using the CUSPARSE API

This section describes how to use the CUSPARSE library API. It does not contain a detailed reference for all API datatypes and functions – those are provided in subsequent chapters. The Legacy CUSPARSE API is also not covered in this section – that is handled in the Appendix A.

2.1 Thread Safety

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

2.2 Scalar Parameters

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

Also, 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. Notice that even though these functions return immediately, similarly to matrix and vector results, the scalar result is ready only when execution of the routine on the GPU completes. This requires proper synchronization in order to read the result from the host.

These changes allow the library functions to execute completely asynchronously using streams even when α and β are generated by a previous kernel. For example, this situation arises when we use the CUSPARSE library 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 each stream with each task. In order to achieve the overlap of computation between the tasks, the user 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, the computation performed in separate streams would be overlapped automatically when possible on the GPU. 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.

We recommend using the new CUSPARSE API with scalar parameters and results passed by reference in the device memory to achieve maximum overlap of the computation when using streams.

Although the user 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 the following dense, sparse vector and 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

3.2.1 Dense Format

Dense vectors are represented with a single data array that is stored linearly in memory. For example, consider the 7×1 dense vector

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.0 & 3.0 & 0.0 & 4.0 \end{bmatrix} \tag{3.1}$$

3.2.2 Sparse Format

Sparse vectors are represented with two arrays.

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

For example, the dense vector (3.1) can be stored as a sparse vector with one-based idexing

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$
$$\begin{bmatrix} 1 & 4 & 5 & 7 \end{bmatrix} \tag{3.2}$$

or as a sparse vector with zero-based idexing

$$\begin{bmatrix}1.0 & 2.0 & 3.0 & 4.0\end{bmatrix}$$
$$\begin{bmatrix}0 & 3 & 4 & 6\end{bmatrix} \tag{3.3}$$

where the top row is the data array and the bottom row is the index array.

Note: It is assumed that the indices are provided in an increasing order and that each index appears only once.

3.3 Matrix Formats

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` $\times$ `n` dense matrix X with leading dimension `ldX` can be stored as

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

with one-based indexing. Its elements are arranged linearly in memory as

$$\left[\; X_{1,1} \quad X_{2,1} \quad \dots \quad X_{m,1} \quad \dots \quad X_{ldX,1} \quad \dots \quad X_{1,n} \quad X_{2,n} \quad \dots \quad X_{m,n} \quad \dots \quad X_{ldX,n} \;\right]$$

Please note that this format and notation is similar to the format and notation used in the NVIDIA CUDA CUBLAS library.

Note: Dense matrices are assumed to be stored in column-major format in memory.

3.3.2 Coordinate Format (COO)

The $\mathtt{m} \times \mathtt{n}$ sparse matrix `A` is represented in COO format by the following parameters:

`nnz`	(integer)	The number of non-zero elements in the matrix.
`cooValA`	(pointer)	Points to the data array of length `nnz` that holds all non-zero 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`.

Sparse matrices are assumed to be stored in row-major COO format, in other words, the index arrays are first sorted by row indices and then within the same row by column indices. Also 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 as

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

and in the COO format with one-based indexing as

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

Note: Sparse matrices in COO format are assumed to be stored in row-major format in memory. Also, it is assumed that each pair of row and column indices appears only once.

3.3.3 Compressed Sparse Row Format (CSR)

The only difference between the COO and CSR formats is that the array containing the row indices is compressed in CSR format. The $m \times n$ sparse matrix A is represented in CSR format by the following parameters:

`nnz`	(integer)	The number of non-zero elements in the matrix.
`csrValA`	(pointer)	Points to the data array of length `nnz` that holds all non-zero 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 non-zero element in the `i`th row for `i=1,...,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 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. Also it is assumed that each pair of row and column indices appears only once.

For example, consider again the 4×5 matrix A:

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

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

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

and in CSR format with one-based indexing as

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

Note: Sparse matrices in CSR format are assumed to be stored in row-major format in memory. Also, it is assumed that each pair of row and column indices appears only once.

3.3.4 Compressed Sparse Column Format (CSC)

The only two differences between the COO and CSC formats is that the matrix is stored in column-major format and that the array containing the column indices is compressed in CSC format. The $\mathtt{m} \times \mathtt{n}$ matrix `A` is represented in CSC format by the following parameters:

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

Note: 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 as

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

and in CSC format with one-based indexing as

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

Note: Sparse matrices in CSC format are assumed to be stored in column-major format in memory. Also, it is assumed that each pair of row and column indices appears only once.

3.3.5 Ellpack-Itpack Format (ELL)

An $\mathtt{m} \times \mathtt{n}$ sparse matrix `A` with at most `k` non-zero elements per row is stored in the Ellpack-Itpack (ELL) format [2] using two dense arrays of dimension $\mathtt{m} \times \mathtt{k}$. The first data array contains the values of the non-zero elements in the matrix, while the second integer array contains the corresponding column indices.

For example, consider the 4×5 matrix `A`:

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

It is stored in ELL format with zero-based indexing as

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

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

and in ELL format with one-based indexing as

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

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

In the above example, if there are less than `k` non-zero elements in a row, we denote the empty spaces in the data array with zero and in the integer array with -1.

This 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.

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

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]. It 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 the user specified criteria.

For more information, please refer to the description of `cusparseHybPartition_t` type, as well as the description of the convertion routines `dense2hyb` and `csr2hyb`.

Note: The sparse matrix in HYB format is always stored using zero-based indexing for both ELL and COO parts.

3.3.7 Block Compressed Sparse Row Format (BSR)

The only difference between the CSR and BSR formats is the format of storage element. The former stores primitive data type (single, double, cuComplex, cuDoubleComplex) whereas the latter stores a 2-dimensional square block of primitive data type. The dimension of the square block is $blockDim$. The $\mathtt{m} \times \mathtt{n}$ 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 non-zero blocks in the matrix.
`bsrValA`	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all elements of non-zero 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 $\mathtt{mb}+1$ that holds indices into the arrays `bsrColIndA` and `bsrValA`. The first `mb` entries of this array contain the indices of the first non-zero block in the `i`th block row for `i=1,...,mb`, while the last entry contains $\mathtt{nnzb}+\mathtt{bsrRowPtrA(0)}$. 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`.

Same as 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 splitted 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

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

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

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

Basic element of BSR is nonzero A_{ij} block which contains at least one nonzero element of A. There are five nonzero blocks among six.

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

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

$$\begin{aligned} \texttt{bsrValA} &= \begin{bmatrix} A_{00} & A_{01} & A_{10} & A_{11} & A_{12} \end{bmatrix} \\ \texttt{bsrRowPtrA} &= \begin{bmatrix} 0 & 2 & 5 \end{bmatrix} \\ \texttt{bsrColIndA} &= \begin{bmatrix} 0 & 1 & 0 & 1 & 2 \end{bmatrix} \end{aligned}$$

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

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

Under row-major order, the physical storage of `bsrValA` is

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

Similarly in BSR format with one-based indexing and column-major order, `A` can be represented by

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

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

$$\begin{aligned} \texttt{bsrRowPtrA} &= \begin{bmatrix} 1 & 3 & 6 \end{bmatrix} \\ \texttt{bsrColIndA} &= \begin{bmatrix} 1 & 2 & 1 & 2 & 3 \end{bmatrix} \end{aligned}$$

Note: storage format of blocks in BSR format can be column-major order or row-major order, independent of base index. However if the user has BSR of MKL and wants to copy BSR of MKL to BSR of CUSPARSE directly, then `cusparseDirection_t` is CUSPARSE_DIRECTION_COLUMN if base index is one. Otherwise, `cusparseDirection_t` is CUSPARSE_DIRECTION_ROW.

3.3.8 Extended BSR Format (BSRX)

The same as BSR format but array `bsrRowPtr` is seperated into two parts, first non-zero block of each row is still specified by the same array `bsrRowPtr`, but the position next to the last non-zero block of each row is specified by array `bsrEndPtr`. Briefly, BSRX format is simply like a 4-vector variant of BSR format.

`A` is represented in BSRX format by the following parameters:

`blockDim`	(integer)	The 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 size of `bsrcolIndA` and `bsrValA`. `nnzb` is greater than or equal to the number of non-zero blocks in the matrix A.
`bsrValA`	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all elements of non-zero 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`. `bsrRowPtr(i)` is the position of the first non-zero block of `i`th block row in the `bsrColIndA` and `bsrValA`.
`bsrEndPtrA`	(pointer)	Points to the integer array of length `mb` that holds indices into the arrays `bsrColIndA` and `bsrValA`. `bsrRowPtr(i)` is the position next to the last non-zero block of `i`th block row in the `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`.

The simple conversion between BSR and BSRX can be done as following. Suppose the user has a 2×3 block sparse matrix A_b represented by

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

and its BSR format is

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

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

$$
\begin{aligned}
\texttt{bsrRowPtrA of BSRX} &= \begin{bmatrix} 0 & 2 \end{bmatrix} \\
\texttt{bsrEndPtrA of BSRX} &= \begin{bmatrix} 2 & 5 \end{bmatrix}
\end{aligned}
$$

The power of BSRX format is that the user can specify submatrix in the original BSR format by modifying `bsrRowPtrA` and `bsrEndPtrA` but keeping `bsrColIndA` and `bsrValA` unchanged.

For example, if the user wants another block matrix $\tilde{A} = \begin{bmatrix} O & O & O \\ O & A_{11} & O \end{bmatrix}$ which is slightly different from A, then he can keep `bsrColIndA` and `bsrValA`, but reconstruct $\tilde{A}$ by proper setting of `bsrRowPtrA` and `bsrEndPtrA`. The following 4-vector characterizes $\tilde{A}$.

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

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 `cusparseCreate()` prior to calling 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 will assume 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 actuall 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.

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 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.

4.10 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_INITIALIZED`	The CUSPARSE library was not initialized. This is usually caused by the lack of a prior `cusparseCreate()` 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_FAILED`	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_VALUE`	An unsupported value or parameter was passed to the function (a negative vector size, for example). **To correct:** ensure that all the parameters being passed have valid values.

`CUSPARSE_STATUS_ARCH_MISMATCH`	The function requires a feature absent from the device architecture; usually caused by the lack of support for atomic operations or double precision. **To correct:** compile and run the application on a device with appropriate compute capability, which is 1.1 for 32-bit atomic operations and 1.3 for double precision.
`CUSPARSE_STATUS_MAPPING_ERROR`	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_EXECUTION_FAILED`	The GPU program failed to execute. This is often caused by a launch failure of the kernel on the GPU, which can be caused by multiple reasons. **To correct:** check that the hardware, an appropriate version of the driver, and the CUSPARSE library are correctly installed.
`CUSPARSE_STATUS_INTERNAL_ERROR`	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_TYPE_NOT_SUPPORTED`	The matrix type is not supported by this function. This is usually caused by passing an invalid matrix descriptor to the function. **To correct:** check that the fields in `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 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.

5.3 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.

5.4 cusparseCreateSolveAnalysisInfo()

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

This function creates and initializes the solve and analysis `info` 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.

5.5 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.

5.6 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.

5.7 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.

5.8 cusparseDestroySolveAnalysisInfo()

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

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

Input

info	the solve and analysis structure.

Status Returned

CUSPARSE_STATUS_SUCCESS	the resources were released successfully.

5.9 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.

5.10 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.

5.11 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.

5.12 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.

5.13 cusparseGetPointerMode()

```
cusparseStatus_t
cusparseGetPointerMode(cusparseHandle_t 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.

5.14 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.

5.15 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 DiagType field was set successfully.
CUSPARSE_STATUS_INVALID_VALUE	An invalid diagType parameter was passed.

5.16 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.

5.17 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.

5.18 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.

5.19 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_NOT_INITIALIZED`	the library was not initialized.

5.20 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_NOT_INITIALIZED`	the library was not initialized.

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 $\mathbf{x}$ in sparse format by the constant α and adds the result to the vector $\mathbf{y}$ in dense format. This operation can be written as

$$\mathbf{y} = \mathbf{y} + \alpha * \mathbf{x},$$

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 non-zero values of vector x.
xInd	integer vector with nnz indices of the non-zero 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 $\mathbf{x}$ in sparse format and vector $\mathbf{y}$ in dense format. This operation can be written as

$$\texttt{result} = \mathbf{y}^T\mathbf{x},$$

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 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 of vector `x`.
`xVal`	<type> vector with `nnz` non-zero values of vector `x`.
`xInd`	integer vector with `nnz` indices of the non-zero 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

$$\texttt{result} = \mathbf{y}^H \mathbf{x},$$

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 it may return control to the application on the host before the result is ready.

Input

`handle`	handle to a CUSPARSE context.
`nnz`	number of elements of vector `x`.
`xVal`	<type> vector with `nnz` non-zero values of vector `x`.
`xInd`	integer vector with `nnz` indices of the non-zero 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 of vector x.
y	<type> vector in dense format (of size ≥ max(xInd)-idxBase+1).
xInd	integer vector with nnz indices of the non-zero values of vector x.
idxBase	CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.

Output

xVal	<type> vector with nnz non-zero 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 zeroes 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 of vector `x`.
`y`	<type> vector in dense format (of size $\geq$ max(`xInd`)-`idxBase+1`).
`xInd`	integer vector with `nnz` indices of non-zero values of vector `x`
`idxBase`	`CUSPARSE_INDEX_BASE_ZERO` or `CUSPARSE_INDEX_BASE_ONE`.

Output

`xVal`	<type> vector with `nnz` non-zero 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 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]
```

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 of vector x.
xVal	<type> vector with nnz non-zero values of vector x.
xInd	integer vector with nnz indices of the non-zero 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 fomat (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 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 of the vector x.
xVal	<type> vector with nnz non-zero values of vector x.
xInd	integer vector with nnz indices of the non-zero 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 non-zero 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

$$\mathbf{y} = \alpha * \mathtt{op}(A) * \mathbf{x} + \beta * \mathbf{y}$$

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

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

Several comments on bsrmv:

1. Only `CUSPARSE_OPERATION_NON_TRANSPOSE` is supported, i.e.

$$\mathbf{y} = \alpha * A * \mathbf{x} + \beta * \mathbf{y}$$

2. Only `CUSPARSE_MATRIX_TYPE_GENERAL` is supported.
3. 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 out-of-array-bound.

Example: suppose the user has a CSR format and wants to try bsrmv, the following code demonstrates csr2csc and csrmv on 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;
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 );
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, 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,
             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 $\texttt{op}(A)$. 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 nonz-zero 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 `nnzb` $(=$ `bsrRowPtr(mb)` $-$ `bsrRowPtr(0)`$)$ non-zero 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 `nnzb` $(=$ `bsrRowPtr(m)` $-$ `bsrRowPtr(0)`$)$ column indices of the non-zero 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 $mb * blockDim$ element.

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, int nb, int nnzb,
   const cuDoubleComplex *alpha, const cusparseMatDescr_t descr,
   const cuDoubleComplex *bsrVal, const int *bsrMaskPtr,
   const int *bsrRowPtr, const int *bsrEndPtr, const int *bsrColInd,
   int blockDim, const cuDoubleComplex *x,
   const cuDoubleComplex *beta, cuDoubleComplex *y)
```

This function performs a bsrmv and a mask operation

$$\mathbf{y}(\mathbf{mask}) = \big(\alpha * \mathtt{op}(A) * \mathbf{x} + \beta * \mathbf{y}\big)(\mathbf{mask})$$

where A is $(mb * blockDim) \times (nb * blockDim)$ sparse matrix (that is defined in BSRX storage format by the four arrays `bsrVal`, `bsrRowPtr`, `bsrEndPtr` and `bsrColInd`), $\mathbf{x}$ and $\mathbf{y}$

are vectors, α and β are scalars, and

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

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

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

$$\begin{aligned} \texttt{bsrVal} &= \begin{bmatrix} A_{11} & A_{12} & A_{21} & A_{22} & A_{23} \end{bmatrix} \\ \texttt{bsrRowPtr} &= \begin{bmatrix} 1 & 3 & 6 \end{bmatrix} \\ \texttt{bsrColInd} &= \begin{bmatrix} 1 & 2 & 1 & 2 & 3 \end{bmatrix} \end{aligned}$$

Suppose we want to do the following bsrmv operation on a matrix $\tilde{A}$ 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 $\tilde{A}$, all that we should do is to keep `bsrVal` and `bsrColInd` unchanged, but modify `bsrRowPtr` and add additional array `bsrEndPtr` which points to last nonzero elements per row of $\tilde{A}$ plus 1.

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

$$\begin{aligned} \texttt{bsrRowPtr} &= \begin{bmatrix} 1 & 4 \end{bmatrix} \\ \texttt{bsrEndPtr} &= \begin{bmatrix} 1 & 5 \end{bmatrix} \end{aligned}$$

Further we can use mask operator (specified by array `bsrMaskPtr`) to update particular row indices of y only because y_1 is never changed. In this case, $\texttt{bsrMaskPtr} = [2]$

The mask operator is equivalent to the following operation (? stands for don't care)

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

In other words, `bsrRowPtr[0]` and `bsrEndPtr[0]` are don't care.

$$\begin{aligned} \texttt{bsrRowPtr} &= \begin{bmatrix} ? & 4 \end{bmatrix} \\ \texttt{bsrEndPtr} &= \begin{bmatrix} ? & 5 \end{bmatrix} \end{aligned}$$

Several comments on bsrxmv:

1. Only `CUSPARSE_OPERATION_NON_TRANSPOSE` and `CUSPARSE_MATRIX_TYPE_GENERAL` are supported.
2. `bsrMaskPtr`, `bsrRowPtr`, `bsrEndPtr` and `bsrColInd` are consistent with base index, either one-based or zero-based. 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 rows of y.
`mb`	number of block rows of matrix A.
`nb`	number of block columns of matrix A.
`nnzb`	size of `bsrColInd`.
`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 `nnzb` non-zero blocks of matrix A.
`bsrMaskPtr`	integer array of `sizOfMask` elements that updated row indices of y.
`bsrRowPtr`	integer array of `mb` elements that contains the start of every block row and the end of the last block row plus one.
`bsrEndPtr`	integer array of `mb` elements that contains the end of every block row plus one.
`bsrColInd`	integer array of `nnzb` column indices of the non-zero 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 $mb * blockDim$ element.

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

$$\mathbf{y} = \alpha * \texttt{op}(A) * \mathbf{x} + \beta * \mathbf{y}$$

where A is $\texttt{m} \times \texttt{n}$ sparse matrix (that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`), $\mathbf{x}$ and $\mathbf{y}$ are vectors, α and β are scalars, and

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

When using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this routine may produce slightly different results during different runs of this function 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 operation different than `CUSPARSE_OPERATION_NON_TRANSPOSE` is selected. 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.
transA	the operation op(A).
m	number of rows of matrix A.
n	number of columns of matrix A.
nnz	number of non-zero elements of 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)) 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 (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero elements of matrix A.
x	<type> vector of n elements if op(A) = A, and m elements if op(A) = A^T or 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 op(A) = A and n elements if op(A) = A^T or 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), symmetric/Hermitian matrix (c.c. >= 1.2) or transpose operation (c.c. >= 1.1).
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.

7.4 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

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

where A is $\texttt{m} \times \texttt{m}$ sparse matrix (that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`), $\mathbf{x}$ and $\mathbf{y}$ are the right-hand-side and the solution vectors, α is a scalar, and

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

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

This function requires 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.
transA	the operation op(A).
m	number of rows of matrix A.
nnz	number of non-zero elements of matrix A.
descrA	the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_TRIANGULAR and diagonal types CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.
csrValA	<type> array of nnz (= csrRowPtrA(m) − csrRowPtrA(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 (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero 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.

7.5 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          *csrValA, const int *csrRowPtrA,
                     const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                     const float          *x, float          *y)
cusparseStatus_t
cusparseDcsrsv_solve(cusparseHandle_t handle, cusparseOperation_t transA, int m,
                     const double         *alpha, const cusparseMatDescr_t descrA,
                     const double         *csrValA, const int *csrRowPtrA,
```

```
                      const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                      const double          *x, double          *y)
cusparseStatus_t
cusparseCcsrsv_solve(cusparseHandle_t handle, cusparseOperation_t transA, int m,
                      const cuComplex       *alpha, const cusparseMatDescr_t descrA,
                      const cuComplex       *csrValA, const int *csrRowPtrA,
                      const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                      const cuComplex       *x, cuComplex       *y)
cusparseStatus_t
cusparseZcsrsv_solve(cusparseHandle_t handle, cusparseOperation_t transA, int m,
                      const cuDoubleComplex *alpha, const cusparseMatDescr_t descrA,
                      const cuDoubleComplex *csrValA, const int *csrRowPtrA,
                      const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                      const cuDoubleComplex *x, cuDoubleComplex *y)
```

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

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

where A is $\mathtt{m} \times \mathtt{m}$ sparse matrix (that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`), $\mathbf{x}$ and $\mathbf{y}$ are the right-hand-side and the solution vectors, α is a scalar, and

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

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 it 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.
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_TRIANGULAR` and diagonal types `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.
`csrValA`	<type> array of `nnz` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(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` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(0)`$)$ column indices of the non-zero 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`.

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`< 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.

7.6 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

$$\mathbf{y} = \alpha * \mathtt{op}(A) * \mathbf{x} + \beta * \mathbf{y}$$

where A is an $\mathtt{m} \times \mathtt{n}$ sparse matrix (that is defined in the HYB storage format by an opaque data structure `hybA`), $\mathbf{x}$ and $\mathbf{y}$ are vectors, α and β are scalars, and

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

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 it 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 $\mathtt{op}(A)$ (currently only $\mathtt{op}(A) = A$ 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.

7.7 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

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

where A is $\texttt{m} \times \texttt{m}$ sparse matrix (that is defined in HYB storage format by an opaque data structure `hybA`), $\mathbf{x}$ and $\mathbf{y}$ are the right-hand-side and the solution vectors, α is a scalar, and

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

Notice that currently only $\texttt{op(A)} = \texttt{A}$ 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 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.
`transA`	the operation op(A) (currently only op(A) = A is supported).
`descrA`	the descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_TRIANGULAR` and diagonal type `CUSPARSE_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.

7.8 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

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

where A is $\texttt{m} \times \texttt{m}$ sparse matrix (that is defined in HYB storage format by an opaque data structure `hybA`), $\mathbf{x}$ and $\mathbf{y}$ are the right-hand-side and the solution vectors, α is a scalar, and

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

Notice that currently only `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 it 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 $\texttt{op}(A)$ (currently only $\texttt{op}(A) = A$ is supported).
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_TRIANGULAR` and diagonal type `CUSPARSE_DIAG_TYPE_NON_UNIT`.
`hybA`	the matrix `A` in HYB storage format.
`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`.

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.

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, int ldb,
               const cuDoubleComplex *beta, cuDoubleComplex *C, int ldc)
```

This function performs one of the following matrix-matrix operation

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

where A is $\texttt{m} \times \texttt{n}$ 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

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

When using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this routine may produce slightly different results during different runs of this function 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 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 operation different than CUSPARSE_OPERATION_NON_TRANSPOSE is selected. 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.
transA	the operation 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 non-zero 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)) 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 (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero elements of matrix A.
B	array of dimensions (ldb, n).
ldb	leading dimension of B. It must be at least max (1, k) if 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 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>csrsm_analysis

```
cusparseStatus_t
cusparseScsrsm_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
cusparseDcsrsm_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
cusparseCcsrsm_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
cusparseZcsrsm_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

$$\texttt{op}(A) * Y = \alpha * X$$

with multiple right-hand-sides, where A is $\texttt{m} \times \texttt{m}$ 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 dense matrices, α is a scalar, and

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

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

This function requires 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.
transA	the operation op(A).
m	number of rows of matrix A.
nnz	number of non-zero elements of matrix A.
descrA	the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_TRIANGULAR and diagonal types CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.
csrValA	<type> array of nnz (= csrRowPtrA(m) − csrRowPtrA(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 (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero 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.

8.3 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           *csrValA, const int *csrRowPtrA,
                     const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                     const float           *X, int ldx,
                     float           *Y, int ldy)
cusparseStatus_t
cusparseDcsrsm_solve(cusparseHandle_t handle, cusparseOperation_t transA,
                     int m, int n, const double *alpha,
                     const cusparseMatDescr_t descrA,
                     const double          *csrValA, const int *csrRowPtrA,
                     const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                     const double          *X, int ldx,
                     double          *Y, int ldy)
cusparseStatus_t
cusparseCcsrsm_solve(cusparseHandle_t handle, cusparseOperation_t transA,
                     int m, int n, const cuComplex *alpha,
                     const cusparseMatDescr_t descrA,
                     const cuComplex       *csrValA, const int *csrRowPtrA,
                     const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                     const cuComplex       *X, int ldx,
                     cuComplex       *Y, int ldy)
cusparseStatus_t
cusparseZcsrsm_solve(cusparseHandle_t handle, cusparseOperation_t transA,
                     int m, int n, const cuDoubleComplex *alpha,
                     const cusparseMatDescr_t descrA,
                     const cuDoubleComplex *csrValA, const int *csrRowPtrA,
                     const int *csrColIndA, cusparseSolveAnalysisInfo_t info,
                     const cuDoubleComplex *X, int ldx,
                     cuDoubleComplex *Y, int ldy)
```

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

$$\texttt{op}(A) * Y = \alpha * X$$

with multiple right-hand-sides, where A is $\texttt{m} \times \texttt{m}$ 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 dense matrices, α is a scalar, and

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

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 it 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 type is `CUSPARSE_MATRIX_TYPE_TRIANGULAR` and diagonal types `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.
`csrValA`	<type> array of `nnz` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(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` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(0)`$)$ column indices of the non-zero 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 array of dimensions `(ldx, n)`.
`ldx`	leading dimension of `X` (that is $\geq \max(1, m)$).

Output

`Y`	<type> solution array of dimensions `(ldy, n)`.
`ldy`	leading dimension of `Y` (that is $\geq \max(1, m)$).

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.

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, const int *csrRowPtrA, const int *csrColIndA,
    const cuComplex *beta,
    const cusparseMatDescr_t descrB, int nnzB,
    const cuComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB,
    const cusparseMatDescr_t descrC,
    cuComplex *csrValC, int *csrRowPtrC, int *csrColIndC)
cusparseStatus_t
cusparseZcsrgeam(cusparseHandle_t handle, int m, int n,
    const cuDoubleComplex *alpha,
    const cusparseMatDescr_t descrA, int nnzA,
    const cuDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA,
    const cuDoubleComplex *beta,
    const cusparseMatDescr_t descrB, int nnzB,
    const cuDoubleComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB,
    const cusparseMatDescr_t descrC,
    cuDoubleComplex *csrValC, int *csrRowPtrC, int *csrColIndC)
```

This function performs following matrix-matrix operation

$$C = \alpha * A + \beta * B$$

where A, B and C are $\mathtt{m} \times \mathtt{n}$ 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 two-step approach to complete sparse matrix C. In the first step, the user allocates `csrRowPtrC` of $(\mathtt{m}+1)$ elements and uses function `cusparseXcsrgeamNnz` to determine `csrRowPtrC` and total number of nonzero elements. In the second step, the user gathers nnzC (number of non-zero 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;
int *nnzTotalDevHostPtr = &nnzC;
// nnzTotalDevHostPtr points to host memory
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:

1. CUSPARSE does not support other three combinations, NT, TN and TT. In order to do any one of above three, the user should use the routine csr2csc to convert $A|B$ to $A^T|B^T$.
2. 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 MatrixType field of descriptor to `CUSPARSE_MATRIX_TYPE_GENERAL`.
3. If the sparsity pattern of matrix C is known, then 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 sparsity patterns. The user can call function cusparseXcsrgeamNnz once to setup sparsity pattern of C, then call function cusparse[S|D|C|Z]geam only for each iteration.
4. The pointers, alpha and beta, must be valid.
5. CUSPARSE would not consider special case when alpha or beta is zero. The sparsity pattern of C is independent of value of alpha and beta. If the user want $C = 0 \times A + 1 \times B^T$, then csr2csc is better than csrgeam.
6. `nnzTotalDevHostPtr` is designed to avoid the user to perform data copy of of `csrRowPtrC(m)` and `csrRowPtrC(0)` explicitly.

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 non-zero elements of sparse matrix A.
csrValA	<type> array of nnzA (= csrRowPtrA(m) − csrRowPtrA(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 nnzA (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero elements of matrix A.
beta	<type> scalar used for multiplication.
descrB	the descriptor of matrix B. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL only.
nnzB	number of non-zero elements of sparse matrix B.
csrValB	<type> array of nnzB (= csrRowPtrB(m) − csrRowPtrB(0)) non-zero 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 non-zero 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)) 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.
csrColIndC	integer array of nnzC (= csrRowPtrC(m) − csrRowPtrC(0)) column indices of the non-zero 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,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>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, cusparseOperation_t transB,
    int m, int n, int k,
    const cusparseMatDescr_t descrA, const int nnzA,
    const cuComplex *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    const cusparseMatDescr_t descrB, const int nnzB,
    const cuComplex *csrValB,
    const int *csrRowPtrB, const int *csrColIndB,
    const cusparseMatDescr_t descrC,
    cuComplex *csrValC,
    const int *csrRowPtrC, int *csrColIndC )
cusparseStatus_t
cusparseZcsrgemm(cusparseHandle_t handle,
    cusparseOperation_t transA, cusparseOperation_t transB,
    int m, int n, int k,
    const cusparseMatDescr_t descrA, const int nnzA,
    const cuDoubleComplex *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    const cusparseMatDescr_t descrB, const int nnzB,
    const cuDoubleComplex *csrValB,
    const int *csrRowPtrB, const int *csrColIndB,
    const cusparseMatDescr_t descrC,
    cuDoubleComplex *csrValC,
    const int *csrRowPtrC, int *csrColIndC )
```

This function performs following matrix-matrix operation

$$\mathbf{C} = \mathtt{op}(A) * \mathtt{op}(B)$$

where $\mathtt{op}(A)$, $\mathtt{op}(B)$ and C are $\mathtt{m} \times \mathtt{k}$, $\mathtt{k} \times \mathtt{n}$, and $\mathtt{m} \times \mathtt{n}$ 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

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

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 $C = A^T * B^T$.

Same as cusparseGeam, CUSPARSE adopts two-step approach to complete sparse matrix C. In the first step, the user allocates `csrRowPtrC` of $(\texttt{m}+1)$ elements and uses function `cusparseXcsrgemmNnz` to determine the number of non-zero elements per row. In the second step, the user gathers nnzC (number of non-zero elements of matrix C) from either $(nnzC = *\texttt{nnzTotalDevHostPtr})$ or $(nnzC = \texttt{csrRowPtrC(m)} - \texttt{csrRowPtrC(0)})$ and allocates `csrValC, 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;
int *nnzTotalDevHostPtr = &nnzC;
// nnzTotalDevHostPtr points to host memory
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgemmNnz(handle, 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:

1. Only NN version is implemented. For NT version, matrix B is converted to B^T by csr2csc and call NN version. The same technique applies to TN and TT. The csr2csc routine would allocate working space implicitly, if the user needs memory management, then NN version is better.
2. NN version needs working space of size `nnzA` integers at least.
3. 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 MatrixType field of descriptor to `CUSPARSE_MATRIX_TYPE_GENERAL`.

4. Only support devices of compute capability 2.0 or above.

Input

`handle`	handle to the CUSPARSE library context.
`transA`	the operation `op`(A).
`transB`	the operation `op`(B).
`m`	number of rows of sparse matrix `op`(A) and `C`.
`n`	number of columns of sparse matrix `op`(B) and `C`.
`k`	number of columns/rows of sparse matrix `op`(A) / `op`(B).
`descrA`	the descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL` only.
`nnzA`	number of non-zero elements of sparse matrix `A`.
`csrValA`	<type> array of `nnzA` non-zero 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` column indices of the non-zero elements of matrix A.
`descrB`	the descriptor of matrix B. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL` only.
`nnzB`	number of non-zero elements of sparse matrix `B`.
`csrValB`	<type> array of `nnzB` non-zero 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 non-zero 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)`) 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.
`csrColIndC`	integer array of `nnzC` (= `csrRowPtrC(m)` − `csrRowPtrC(0)`) column indices of the non-zero 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, or IndexBase of descrA,descrB,descrC is not base-0 or base-1).
CUSPARSE_STATUS_ARCH_MISMATCH	only supports compute capability 2.0 or above.
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 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. While 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 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

$$op(A) \approx R^T R$$

where A is $\texttt{m} \times \texttt{m}$ Hermitian/symmetric positive definite sparse matrix (that is defined in CSR storage format by the three arrays `csrValM`, `csrRowPtrA` and `csrColIndA`) and

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

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 factor R^T or R, respectively.

A call to this routine must be preceeded by a call to the `csrsv_analysis` routine.

This function requires some 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.
`transA`	the operation $\texttt{op}(A)$.
`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)`) non-zero elements of matrix A.
`csrRowPtrA`	integer array of $\texttt{m}+1$ 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 non-zero 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 containg 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.

10.2 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

$$op(A) \approx LU$$

where A is $\mathtt{m} \times \mathtt{m}$ sparse matrix (that is defined in CSR storage format by the three arrays `csrValM`, `csrRowPtrA` and `csrColIndA`) and

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

Notice that the diagonal of lower triangular factor L is unitary and need not be stored. Therefore the input matrix is ovewritten with the resulting lower and upper triangular factor L and U, respectively.

A call to this routine must be preceeded by a call to the `csrsv_analysis` routine.

This function requires some 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.
`transA`	the operation $\mathtt{op}(A)$.
`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)`) 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` (= `csrRowPtrA(m)` − `csrRowPtrA(0)`) column indices of the non-zero 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 containg 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.

10.3 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

$$A * Y = \alpha * X$$

with multiple right-hand-sides.

The coefficient matrix A of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**ld**), main (**d**) and upper (**ud**) matrix diagonals, while the right-hand-sides are stored in the dense matrix X. Notice that the solutions Y overwrite the right-hand-sides X on exit.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when `m` is a power of 2.

This routine requires significant amount of temporary extra storage ($\mathtt{m} \times (3 + \mathtt{n})$ $\times$`sizeof(<type>)`). 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	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, m).
ldb	leading dimension of B (that is $\geq \max(1, m)$).

Output

B	<type> dense solution array of dimensions (ldb, m).

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.

10.4 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

$$A^{(i)} * \mathbf{y}^{(i)} = \alpha * \mathbf{x}^{(i)}$$

for $i = 0, \ldots,$ `batchCount`.

The coefficient matrix A of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**ld**), main (**d**) and upper (**ud**) matrix diagonals, while the right-hand-side is stored in the vector **x**. Notice that the solution **y** overwrites the right-hand-side **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 Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when `m` is a power of 2.

This routine requires significant amount of temporary extra storage (`batchCount` $\times(4 \times$ `m` $+ 2048) \times$ `sizeof(<type>)`). 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`	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 $\mathbf{dl}^{(i)}$ that corresponds to the ith linear system starts at location `dl` $+$ `batchStride` $\times$ `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 $\mathbf{d}^{(i)}$ that corresponds to the ith linear system starts at location `d` $+$ `batchStride` $\times$ `i` in memory.
`du`	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $\mathbf{du}^{(i)}$ that corresponds to the ith linear system starts at location `du` $+$ `batchStride` $\times$ `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 $\mathbf{x}^{(i)}$ that corresponds to the ith linear system starts at location `x` $+$ `batchStride` $\times$ `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 $\mathbf{x}^{(i)}$ that corresponds to the ith linear system starts at location $\texttt{x} + \texttt{batchStride} \times \texttt{i}$ 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.

Chapter 11
CUSPARSE Format Conversion Reference

This chapter describes the conversion routines between different sparse and dense storage formats.

11.1 cusparse<t>bsr2csr

```
cusparseStatus_t
cusparseSbsr2csr(cusparseHandle_t handle, cusparseDirection_t dirA,
     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 dirA,
      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 dirA,
      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 dirA,
```

```
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 number of rows of A and $n(= nb * blockDim)$ be number of columns of A, then A and C are $\texttt{m} \times \texttt{n}$ sparse matricies. BSR format of A contains $nnzb(= \texttt{csrRowPtrC(mb)} - \texttt{csrRowPtrC(0)})$ non-zero blocks whereas sparse matrix A contains $nnz(= nnzb * blockDim^2)$ elements. The user must allocate enough space for arrays `csrRowPtrC`, `csrColIndC` and `csrValC`. The requirements are

1. `csrRowPtrC` of $\texttt{m} + 1$ elements,
2. `csrValC` of nnz elements, and
3. `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 dirA = 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, dirA, mb, nb,
        descrA,
        bsrValA, bsrRowPtrA, bsrColIndA,
        blockDim,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

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 of sparse matrix A. The number of rows of sparse matrix C is $m(= mb * blockDim)$.
nb	number of block columns of sparse matrix A. The number of columns of sparse matrix C is $n(= nb * blockDim)$.
descrA	the descriptor of matrix A.
bsrValA	<type> array of nnzb*$blockDim^2$ 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.
bsrColIndA	integer array of nnzb column indices of the non-zero blocks of matrix A.
blockDim	block dimension of sparse matrix A, larger than zero.
descrC	the descriptor of matrix C.

Output

csrValC	<type> array of nnz (= csrRowPtrC(m) − csrRowPtrC(0)) 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.
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_INVALID_VALUE	invalid parameters were passed (mb,nb < 0, IndexBase of descrA,descrC is not base-0 or base-1, dirA 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.

11.2 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 it 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 $\texttt{m}+1$ 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.

11.3 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 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`.
`cscValA`	<type> array of `nnz` (= `cscColPtrA(m)` − `cscColPtrA(0)`) non-zero elements of matrix A.
`cscRowIndA`	integer array of `nnz` (= `cscColPtrA(m)` − `cscColPtrA(0)`) row indices of the non-zero 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.
`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.

11.4 cusparse<t>csr2bsr

```
cusparseStatus_t
cusparseXcsr2bsrNnz(cusparseHandle_t handle, cusparseDirection_t dirA,
      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 dirA,
      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 dirA,
      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 dirA,
      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 dirA,
      int m, int n,
      const cusparseMatDescr_t descrA, const cuDoubleComplex *csrValA,
      const int *csrRowPtrA, const int *csrColIndA,
      int blockDim,
      const cusparseMatDescr_t descrC,
      cuDoubleComplex *bsrValC, int *bsrRowPtrC, int *bsrColIndC)
```

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 $\mathtt{m} \times \mathtt{n}$ sparse matrix and C is $(\mathtt{mb}*\text{blockDim}) \times (\mathtt{nb}*\text{blockDim})$ sparse matrix.

where mb $(=\frac{m+blockDim-1}{blockDim})$ is number of block rows of A and nb $(=\frac{n+blockDim-1}{blockDim})$ is number of block columns of A. m and n need not be multiple of $blockDim$. If so, then zeros are filled in.

CUSPARSE adopts two-step approach to do the conversion. First, the user allocates `bsrRowPtrC` of $(\mathtt{mb}+1)$ elements and uses function `cusparseXcsr2bsrNnz` to determine number of non-zero block columns per block row. Second, the user gathers nnzb (number of non-zero block columns of matrix A) from either $(nnzb = *nnzTotalDevHostPtr)$ or `bsrRowPtrC` $(nnzb = \mathtt{bsrRowPtrC(mb)} - \mathtt{bsrRowPtrC(0)})$ and allocates `bsrValC` of $nnzb * blockDim^2$ elements and `bsrColIndC` of $nnzb$ elements. Finally function cusparse[S|D|C|Z]csr2bsr 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 dirA = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + blockDim-1)/blockDim;
int *nnzTotalDevHostPtr = &nnzb;
// nnzTotalDevHostPtr points to host memory
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
cusparseXcsr2bsrNnz(handle, dirA, 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, dirA, m, n,
        descrA,
        csrValA, csrRowPtrA, csrColIndA,
        blockDim,
        descrC,
        bsrValC, bsrRowPtrC, bsrColIndC);
```

If $blockDim$ is large (typically a block cannot fit into shared memory), then csr2bsr routines will allocate temporary integer array of size $mb * blockDim$. If device memory is not available, then CUSPARSE_STATUS_ALLOC_FAILED is returned.

Input

handle	handle to the CUSPARSE library context.
dirA	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.
nnzA	number of non-zero elements of sparse matrix A.
csrValA	<type> array of nnz $(=$ csrRowPtrA(m) $-$ csrRowPtrA(0)$)$ non-zero elements of matrix A.
csrRowPtrA	integer array of $\mathtt{m}+1$ 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 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$*blockDim^2$ non-zero elements of matrix C.
bsrRowPtrC	integer array of $\mathtt{mb}+1$ elements that contains the start of every block row and the end of the last block row plus one.
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, dirA 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_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.

11.5 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 it 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 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

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.

11.6 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 `csrValA`, `csrRowPtrA` and `csrColIndA`) 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 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 non-zero elements of matrix `A`.
`csrValA`	<type> array of `nnz` (= `csrRowPtrA(m)` − `csrRowPtrA(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` (= `csrRowPtrA(m)` − `csrRowPtrA(0)`) column indices of the non-zero elements of matrix A.
`copyValues`	`CUSPARSE_ACTION_SYMBOLIC` or `CUSPARSE_ACTION_NUMERIC`.
`idxBase`	`CUSPARSE_INDEX_BASE_ZERO` or `CUSPARSE_INDEX_BASE_ONE`.

Output

cscValA	<type> array of nnz (= cscRowPtrA(m) − cscRowPtrA(0)) non-zero 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)) column indices of the non-zero 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_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.

11.7 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)`$)$ non-zero elements of matrix A.
`csrRowPtrA`	integer array of $\texttt{m}+1$ 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 non-zero 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.

11.8 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 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 in CSR format. 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)`) 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` (= `csrRowPtrA(m)` − `csrRowPtrA(0)`) column indices of the non-zero 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 non-zeros 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` on page 18 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.

11.9 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 pre-computed with `cusparse<t>nnz()`.

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.
A	array of dimensions (lda, n).
lda	leading dimension of dense array A.
nnzPerCol	array of size n containing the number of non-zero elements per column.

Output

cscValA	<type> array of nnz (= cscColPtrA(m) − cscColPtrA(0)) non-zero elements of matrix A.
cscRowIndA	integer array of nnz (= cscColPtrA(m) − cscColPtrA(0)) row indices of the non-zero 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.

11.10 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 the `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 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`.
`A`	array of dimensions `(lda, n)`.
`lda`	leading dimension of dense array `A`.
`nnzPerRow`	array of size `m` containing the number of non-zero elements per row.

Output

`csrValA`	<type> array of `nnz` (= `csrRowPtrA(m)` − `csrRowPtrA(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` (= `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_INTERNAL_ERROR`	an internal operation failed.
`CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED`	the matrix type is not supported.

11.11 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 the 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 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 the dense 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 non-zero elements per row.
`userEllWidth`	width of the regular (ELL) part of the matrix in HYB format, which should be less than maximum number of non-zeros 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` on page 18 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.

11.12 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 it 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 the 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)$)$ non-zero elements of matrix A.
csrRowPtrA	integer array of $\texttt{m}+1$ 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 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.

11.13 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 hybA) into a 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.
`descrA`	the descriptor of the 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.

11.14 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 non-zero elements per row or column and the total number of non-zero elements in a dense matrix.

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.
`dirA`	direction that specifies whether to count non-zero elements by `CUSPARSE_DIRECTION_ROW` or `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 non-zero elements per row or column, respectively.
`nnzTotalDevHostPtr`	total number of non-zero 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.

Chapter 12

Appendix A: Using the CUSPARSE Legacy API

This appendix does not provide a full reference of each Legacy API datatype and entry point. Instead, it describes how to use the API, especially where this is different from the regular CUSPARSE API.

Note that in this section, all references to the "CUSPARSE Library" refer to the Legacy CUSPARSE API only.

12.1 Thread Safety

The legacy API is also thread safe when used with multiple host threads and devices.

12.2 Scalar Parameters

In the legacy CUSPARSE API, scalar parameters are passed by value from the host. Also, the few functions that do return a scalar result, such as `doti()` and `nnz()`, return the resulting value on the host, and hence these routines will wait for kernel execution on the device to complete before returning, which makes parallelism with streams impractical. However, the majority of functions do not return any value, in order to be more compatible with Fortran and the existing sparse libraries.

12.3 Helper Functions

In this section we list the helper functions provided by the legacy CUSPARSE API and their functionality. For the exact prototypes of these functions please refer to the legacy CUSPARSE API header file "cusparse.h".

Helper function	Meaning
`cusparseSetKernelStream()`	sets the stream to be used by the library

12.4 Level-1,2,3 Functions

The Level-1,2,3 CUSPARSE functions (also called core functions) have the same name and behavior as the ones listed in the chapters 6, 7 and 8 in this document. Notice that not all of the routines are available in the legacy API. Please refer to the legacy CUSPARSE API header file "cusparse.h" for their exact prototype. Also, the next section talks a bit more about the differences between the legacy and the CUSPARSE API prototypes, and more specifically how to convert the function calls from one API to another.

12.5 Converting Legacy to the CUSPARSE API

There are a few general rules that can be used to convert from legacy to the CUSPARSE API.

1. Exchange the header file "cusparse.h" for "cusparse_v2.h".
2. Exchange the function `cusparseSetKernelStream()` for `cusparseSetStream()`.
3. Change the scalar parameters to be passed by reference, instead of by value (usually simply adding "&" symbol in C/C++ is enough, because the parameters are passed by reference on the host by *default*). However, note that if the routine is running asynchronously, then the variable holding the scalar parameter cannot be changed until the kernels that the routine dispatches are completed. In order to improve parallelism with streams, please refer to the sections 2.2 and 2.3 of this document. Also, see the *NVIDIA CUDA C Programming Guide* for a detailed discussion of how to use streams.
4. Add the parameter "int nnz"as the 5th, 4th, 6th and 4th parameter in the routines `csrmv`, `csrsv_analysis`, `csrmm` and `csr2csc`, respectively. If this parameter is not available explicitly, it can be obtained using the following piece of code

```
cudaError_t cudaStat;
int nnz;

cudaStat = cudaMemcpy(&nnz, &csrRowPtrA[m], (size_t)sizeof(int),
           cudaMemcpyDeviceToHost);
if (cudaStat != cudaSuccess){
    return CUSPARSE_STATUS_INTERNAL_ERROR;
}
if (cusparseGetMatIndexBase(descrA) == CUSPARSE_INDEX_BASE_ONE){
    nnz = nnz-1;
}
```

5. Change the 10th parameter to the function `csr2csc` from `int` 0 or 1 to `enum CUSPARSE_ACTION_SYMBOLIC` or `CUSPARSE_ACTION_NUMERIC`, respectively.

Finally, please use the function prototypes in the header files "cusparse.h" and "cusparse_v2.h" to check the code for correctness.

Chapter 13

Appendix B: 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_v2.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;
    cooRowIndexHostPtr[3]=1; cooColIndexHostPtr[3]=1; cooValHostPtr[3]=4.0;
    cooRowIndexHostPtr[4]=2; cooColIndexHostPtr[4]=0; cooValHostPtr[4]=5.0;
    cooRowIndexHostPtr[5]=2; cooColIndexHostPtr[5]=2; cooValHostPtr[5]=6.0;
    cooRowIndexHostPtr[6]=2; cooColIndexHostPtr[6]=3; cooValHostPtr[6]=7.0;
    cooRowIndexHostPtr[7]=3; cooColIndexHostPtr[7]=1; cooValHostPtr[7]=8.0;
    cooRowIndexHostPtr[8]=3; cooColIndexHostPtr[8]=3; cooValHostPtr[8]=9.0;
```

```
    /*
    //print the matrix
    printf("Input data:\n");
    for (int i=0; i<nnz; i++){
        printf("cooRowIndexHostPtr[%d]=%d  ",i,cooRowIndexHostPtr[i]);
        printf("cooColIndexHostPtr[%d]=%d  ",i,cooColIndexHostPtr[i]);
        printf("cooValHostPtr[%d]=%f     \n",i,cooValHostPtr[i]);
    }
    */

    /* create a sparse and dense vector */
    /* xVal= [100.0 200.0 400.0]   (sparse)
       xInd= [0     1     3    ]
       y   = [10.0 20.0 30.0 40.0 | 50.0 60.0 70.0 80.0] (dense) */
    nnz_vector = 3;
    xIndHostPtr = (int *)   malloc(nnz_vector*sizeof(xIndHostPtr[0]));
    xValHostPtr = (double *)malloc(nnz_vector*sizeof(xValHostPtr[0]));
    yHostPtr    = (double *)malloc(2*n       *sizeof(yHostPtr[0]));
    zHostPtr    = (double *)malloc(2*(n+1)   *sizeof(zHostPtr[0]));
    if((!xIndHostPtr) || (!xValHostPtr) || (!yHostPtr) || (!zHostPtr)){
        CLEANUP("Host malloc failed (vectors)");
        return 1;
    }
    yHostPtr[0] = 10.0;  xIndHostPtr[0]=0; xValHostPtr[0]=100.0;
    yHostPtr[1] = 20.0;  xIndHostPtr[1]=1; xValHostPtr[1]=200.0;
    yHostPtr[2] = 30.0;
    yHostPtr[3] = 40.0;  xIndHostPtr[2]=3; xValHostPtr[2]=400.0;
    yHostPtr[4] = 50.0;
    yHostPtr[5] = 60.0;
    yHostPtr[6] = 70.0;
    yHostPtr[7] = 80.0;
    /*
    //print the vectors
    for (int j=0; j<2; j++){
        for (int i=0; i<n; i++){
            printf("yHostPtr[%d,%d]=%f\n",i,j,yHostPtr[i+n*j]);
        }
    }
    for (int i=0; i<nnz_vector; i++){
        printf("xIndHostPtr[%d]=%d  ",i,xIndHostPtr[i]);
        printf("xValHostPtr[%d]=%f\n",i,xValHostPtr[i]);
    }
    */

    /* allocate GPU memory and copy the matrix and vectors into it */
    cudaStat1 = cudaMalloc((void**)&cooRowIndex,nnz*sizeof(cooRowIndex[0]));
    cudaStat2 = cudaMalloc((void**)&cooColIndex,nnz*sizeof(cooColIndex[0]));
    cudaStat3 = cudaMalloc((void**)&cooVal,     nnz*sizeof(cooVal[0]));
    cudaStat4 = cudaMalloc((void**)&y,          2*n*sizeof(y[0]));
    cudaStat5 = cudaMalloc((void**)&xInd,nnz_vector*sizeof(xInd[0]));
    cudaStat6 = cudaMalloc((void**)&xVal,nnz_vector*sizeof(xVal[0]));
    if ((cudaStat1 != cudaSuccess) ||
        (cudaStat2 != cudaSuccess) ||
        (cudaStat3 != cudaSuccess) ||
        (cudaStat4 != cudaSuccess) ||
        (cudaStat5 != cudaSuccess) ||
        (cudaStat6 != cudaSuccess)) {
        CLEANUP("Device malloc failed");
        return 1;
    }
    cudaStat1 = cudaMemcpy(cooRowIndex, cooRowIndexHostPtr,
                           (size_t)(nnz*sizeof(cooRowIndex[0])),
```

```
                              cudaMemcpyHostToDevice);
    cudaStat2 = cudaMemcpy(cooColIndex, cooColIndexHostPtr,
                              (size_t)(nnz*sizeof(cooColIndex[0])),
                              cudaMemcpyHostToDevice);
    cudaStat3 = cudaMemcpy(cooVal,      cooValHostPtr,
                              (size_t)(nnz*sizeof(cooVal[0])),
                              cudaMemcpyHostToDevice);
    cudaStat4 = cudaMemcpy(y,           yHostPtr,
                              (size_t)(2*n*sizeof(y[0])),
                              cudaMemcpyHostToDevice);
    cudaStat5 = cudaMemcpy(xInd,        xIndHostPtr,
                              (size_t)(nnz_vector*sizeof(xInd[0])),
                              cudaMemcpyHostToDevice);
    cudaStat6 = cudaMemcpy(xVal,        xValHostPtr,
                              (size_t)(nnz_vector*sizeof(xVal[0])),
                              cudaMemcpyHostToDevice);
    if ((cudaStat1 != cudaSuccess) ||
        (cudaStat2 != cudaSuccess) ||
        (cudaStat3 != cudaSuccess) ||
        (cudaStat4 != cudaSuccess) ||
        (cudaStat5 != cudaSuccess) ||
        (cudaStat6 != cudaSuccess)) {
        CLEANUP("Memcpy from Host to Device failed");
        return 1;
    }

    /* initialize cusparse library */
    status= cusparseCreate(&handle);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("CUSPARSE Library initialization failed");
        return 1;
    }

    /* create and setup matrix descriptor */
    status= cusparseCreateMatDescr(&descr);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix descriptor initialization failed");
        return 1;
    }
    cusparseSetMatType(descr,CUSPARSE_MATRIX_TYPE_GENERAL);
    cusparseSetMatIndexBase(descr,CUSPARSE_INDEX_BASE_ZERO);

    /* exercise conversion routines (convert matrix from COO 2 CSR format) */
    cudaStat1 = cudaMalloc((void**)&csrRowPtr,(n+1)*sizeof(csrRowPtr[0]));
    if (cudaStat1 != cudaSuccess) {
        CLEANUP("Device malloc failed (csrRowPtr)");
        return 1;
    }
    status= cusparseXcoo2csr(handle,cooRowIndex,nnz,n,
                                csrRowPtr,CUSPARSE_INDEX_BASE_ZERO);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Conversion from COO to CSR format failed");
        return 1;
    }
    //csrRowPtr = [0 3 4 7 9]

    /* exercise Level 1 routines (scatter vector elements) */
    status= cusparseDsctr(handle, nnz_vector, xVal, xInd,
                          &y[n], CUSPARSE_INDEX_BASE_ZERO);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Scatter from sparse to dense vector failed");
        return 1;
```

```
    }
    //y = [10 20 30 40 | 100 200 70 400]

    /* exercise Level 2 routines (csrmv) */
    status= cusparseDcsrmv(handle,CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, nnz,
                           &dtwo, descr, cooVal, csrRowPtr, cooColIndex,
                           &y[0], &dthree, &y[n]);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix-vector multiplication failed");
        return 1;
    }
    //y = [10 20 30 40 | 680 760 1230 2240]
    cudaMemcpy(yHostPtr, y, (size_t)(2*n*sizeof(y[0])), cudaMemcpyDeviceToHost);
    /*
    printf("Intermediate results:\n");
    for (int j=0; j<2; j++){
        for (int i=0; i<n; i++){
            printf("yHostPtr[%d,%d]=%f\n",i,j,yHostPtr[i+n*j]);
        }
    }
    */

    /* exercise Level 3 routines (csrmm) */
    cudaStat1 = cudaMalloc((void**)&z, 2*(n+1)*sizeof(z[0]));
    if (cudaStat1 != cudaSuccess) {
        CLEANUP("Device malloc failed (z)");
        return 1;
    }
    cudaStat1 = cudaMemset((void *)z,0, 2*(n+1)*sizeof(z[0]));
    if (cudaStat1 != cudaSuccess) {
        CLEANUP("Memset on Device failed");
        return 1;
    }
    status= cusparseDcsrmm(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, 2, n,
                           nnz, &dfive, descr, cooVal, csrRowPtr, cooColIndex,
                           y, n, &dzero, z, n+1);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix-matrix multiplication failed");
        return 1;
    }

    /* print final results (z) */
    cudaStat1 = cudaMemcpy(zHostPtr, z,
                           (size_t)(2*(n+1)*sizeof(z[0])),
                           cudaMemcpyDeviceToHost);
    if (cudaStat1 != cudaSuccess)  {
        CLEANUP("Memcpy from Device to Host failed");
        return 1;
    }
    //z = [950 400 2550 2600 0 | 49300 15200 132300 131200 0]
    /*
    printf("Final results:\n");
    for (int j=0; j<2; j++){
        for (int i=0; i<n+1; i++){
            printf("z[%d,%d]=%f\n",i,j,zHostPtr[i+(n+1)*j]);
        }
    }
    */

    /* destroy matrix descriptor */
    status = cusparseDestroyMatDescr(descr);
    descr = 0;
```

```
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix descriptor destruction failed");
        return 1;
    }

    /* destroy handle */
    status = cusparseDestroy(handle);
    handle = 0;
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("CUSPARSE Library release of resources failed");
        return 1;
    }

    /* check the results */
    /* Notice that CLEANUP() contains a call to cusparseDestroy(handle) */
    if ((zHostPtr[0] != 950.0)     ||
        (zHostPtr[1] != 400.0)     ||
        (zHostPtr[2] != 2550.0)    ||
        (zHostPtr[3] != 2600.0)    ||
        (zHostPtr[4] != 0.0)       ||
        (zHostPtr[5] != 49300.0)   ||
        (zHostPtr[6] != 15200.0)   ||
        (zHostPtr[7] != 132300.0)  ||
        (zHostPtr[8] != 131200.0)  ||
        (zHostPtr[9] != 0.0)       ||
        (yHostPtr[0] != 10.0)      ||
        (yHostPtr[1] != 20.0)      ||
        (yHostPtr[2] != 30.0)      ||
        (yHostPtr[3] != 40.0)      ||
        (yHostPtr[4] != 680.0)     ||
        (yHostPtr[5] != 760.0)     ||
        (yHostPtr[6] != 1230.0)    ||
        (yHostPtr[7] != 2240.0)){
        CLEANUP("example test FAILED");
        return 1;
    }
    else{
        CLEANUP("example test PASSED");
        return 0;
    }
}
```

Chapter 14
Appendix C: 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'`.

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"
c     predefined constants (need to be careful with them)
      dzero = 0.0
      dtwo  = 2.0
      dthree= 3.0
      dfive = 5.0
c     create the following sparse test matrix in COO format
c     (notice one-based indexing)
c     |1.0     2.0 3.0|
c     |    4.0        |
c     |5.0     6.0 7.0|
c     |    8.0     9.0|
      cooRowIndexHostPtr(1)=1
      cooColIndexHostPtr(1)=1
      cooValHostPtr(1)     =1.0
      cooRowIndexHostPtr(2)=1
      cooColIndexHostPtr(2)=3
      cooValHostPtr(2)     =2.0
      cooRowIndexHostPtr(3)=1
      cooColIndexHostPtr(3)=4
      cooValHostPtr(3)     =3.0
      cooRowIndexHostPtr(4)=2
      cooColIndexHostPtr(4)=2
      cooValHostPtr(4)     =4.0
      cooRowIndexHostPtr(5)=3
      cooColIndexHostPtr(5)=1
      cooValHostPtr(5)     =5.0
      cooRowIndexHostPtr(6)=3
      cooColIndexHostPtr(6)=3
      cooValHostPtr(6)     =6.0
      cooRowIndexHostPtr(7)=3
      cooColIndexHostPtr(7)=4
      cooValHostPtr(7)     =7.0
```

```
      cooRowIndexHostPtr(8)=4
      cooColIndexHostPtr(8)=2
      cooValHostPtr(8)     =8.0
      cooRowIndexHostPtr(9)=4
      cooColIndexHostPtr(9)=4
      cooValHostPtr(9)     =9.0
c     print the matrix
      write(*,*) "Input data:"
      do i=1,nnz
         write(*,*) "cooRowIndexHostPtr[",i,"]=",cooRowIndexHostPtr(i)
         write(*,*) "cooColIndexHostPtr[",i,"]=",cooColIndexHostPtr(i)
         write(*,*) "cooValHostPtr[",     i,"]=",cooValHostPtr(i)
      enddo

c     create a sparse and dense vector
c     xVal= [100.0 200.0 400.0]   (sparse)
c     xInd= [0     1     3    ]
c     y   = [10.0 20.0 30.0 40.0 | 50.0 60.0 70.0 80.0] (dense)
c     (notice one-based indexing)
      yHostPtr(1) = 10.0
      yHostPtr(2) = 20.0
      yHostPtr(3) = 30.0
      yHostPtr(4) = 40.0
      yHostPtr(5) = 50.0
      yHostPtr(6) = 60.0
      yHostPtr(7) = 70.0
      yHostPtr(8) = 80.0
      xIndHostPtr(1)=1
      xValHostPtr(1)=100.0
      xIndHostPtr(2)=2
      xValHostPtr(2)=200.0
      xIndHostPtr(3)=4
      xValHostPtr(3)=400.0
c     print the vectors
      do j=1,2
         do i=1,n
            write(*,*) "yHostPtr[",i,",",j,"]=",yHostPtr(i+n*(j-1))
         enddo
      enddo
      do i=1,nnz_vector
         write(*,*) "xIndHostPtr[",i,"]=",xIndHostPtr(i)
         write(*,*) "xValHostPtr[",i,"]=",xValHostPtr(i)
      enddo

c     allocate GPU memory and copy the matrix and vectors into it
c     cudaSuccess=0
c     cudaMemcpyHostToDevice=1
      cudaStat1 = cuda_malloc(cooRowIndex,nnz*4)
      cudaStat2 = cuda_malloc(cooColIndex,nnz*4)
      cudaStat3 = cuda_malloc(cooVal,     nnz*8)
      cudaStat4 = cuda_malloc(y,          2*n*8)
      cudaStat5 = cuda_malloc(xInd,nnz_vector*4)
      cudaStat6 = cuda_malloc(xVal,nnz_vector*8)
      if ((cudaStat1 /= 0) .OR.
     $    (cudaStat2 /= 0) .OR.
     $    (cudaStat3 /= 0) .OR.
     $    (cudaStat4 /= 0) .OR.
     $    (cudaStat5 /= 0) .OR.
     $    (cudaStat6 /= 0)) then
         write(*,*) "Device malloc failed"
         write(*,*) "cudaStat1=",cudaStat1
         write(*,*) "cudaStat2=",cudaStat2
```

```
           write(*,*) "cudaStat3=",cudaStat3
           write(*,*) "cudaStat4=",cudaStat4
           write(*,*) "cudaStat5=",cudaStat5
           write(*,*) "cudaStat6=",cudaStat6
           stop
        endif
        cudaStat1 = cuda_memcpy_fort2c_int(cooRowIndex,cooRowIndexHostPtr,
     $                                    nnz*4,1)
        cudaStat2 = cuda_memcpy_fort2c_int(cooColIndex,cooColIndexHostPtr,
     $                                    nnz*4,1)
        cudaStat3 = cuda_memcpy_fort2c_real(cooVal,    cooValHostPtr,
     $                                     nnz*8,1)
        cudaStat4 = cuda_memcpy_fort2c_real(y,       yHostPtr,
     $                                     2*n*8,1)
        cudaStat5 = cuda_memcpy_fort2c_int(xInd,       xIndHostPtr,
     $                                    nnz_vector*4,1)
        cudaStat6 = cuda_memcpy_fort2c_real(xVal,      xValHostPtr,
     $                                     nnz_vector*8,1)
        if ((cudaStat1 /= 0) .OR.
     $      (cudaStat2 /= 0) .OR.
     $      (cudaStat3 /= 0) .OR.
     $      (cudaStat4 /= 0) .OR.
     $      (cudaStat5 /= 0) .OR.
     $      (cudaStat6 /= 0)) then
           write(*,*) "Memcpy from Host to Device failed"
           write(*,*) "cudaStat1=",cudaStat1
           write(*,*) "cudaStat2=",cudaStat2
           write(*,*) "cudaStat3=",cudaStat3
           write(*,*) "cudaStat4=",cudaStat4
           write(*,*) "cudaStat5=",cudaStat5
           write(*,*) "cudaStat6=",cudaStat6
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
           call cuda_free(xVal)
           call cuda_free(y)
           stop
        endif

c       initialize cusparse library
c       CUSPARSE_STATUS_SUCCESS=0
        status = cusparse_create(handle)
        if (status /= 0) then
           write(*,*) "CUSPARSE Library initialization failed"
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
           call cuda_free(xVal)
           call cuda_free(y)
           stop
        endif
c       get version
c       CUSPARSE_STATUS_SUCCESS=0
        status = cusparse_get_version(handle,version)
        if (status /= 0) then
           write(*,*) "CUSPARSE Library initialization failed"
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
```

```
          call cuda_free(xVal)
          call cuda_free(y)
          call cusparse_destroy(handle)
          stop
       endif
       write(*,*) "CUSPARSE Library version",version

c      create and setup the matrix descriptor
c      CUSPARSE_STATUS_SUCCESS=0
c      CUSPARSE_MATRIX_TYPE_GENERAL=0
c      CUSPARSE_INDEX_BASE_ONE=1
       status= cusparse_create_mat_descr(descrA)
       if (status /= 0) then
          write(*,*) "Creating matrix descriptor failed"
          call cuda_free(cooRowIndex)
          call cuda_free(cooColIndex)
          call cuda_free(cooVal)
          call cuda_free(xInd)
          call cuda_free(xVal)
          call cuda_free(y)
          call cusparse_destroy(handle)
          stop
       endif
       status = cusparse_set_mat_type(descrA,0)
       status = cusparse_set_mat_index_base(descrA,1)
c      print the matrix descriptor
       mtype = cusparse_get_mat_type(descrA)
       fmode = cusparse_get_mat_fill_mode(descrA)
       dtype = cusparse_get_mat_diag_type(descrA)
       ibase = cusparse_get_mat_index_base(descrA)
       write (*,*) "matrix descriptor:"
       write (*,*) "t=",mtype,"m=",fmode,"d=",dtype,"b=",ibase

c      exercise conversion routines (convert matrix from COO 2 CSR format)
c      cudaSuccess=0
c      CUSPARSE_STATUS_SUCCESS=0
c      CUSPARSE_INDEX_BASE_ONE=1
       cudaStat1 = cuda_malloc(csrRowPtr,(n+1)*4)
       if (cudaStat1 /= 0) then
          call cuda_free(cooRowIndex)
          call cuda_free(cooColIndex)
          call cuda_free(cooVal)
          call cuda_free(xInd)
          call cuda_free(xVal)
          call cuda_free(y)
          call cusparse_destroy_mat_descr(descrA)
          call cusparse_destroy(handle)
          write(*,*) "Device malloc failed (csrRowPtr)"
          stop
       endif
       status= cusparse_xcoo2csr(handle,cooRowIndex,nnz,n,
     $                           csrRowPtr,1)
       if (status /= 0) then
          call cuda_free(cooRowIndex)
          call cuda_free(cooColIndex)
          call cuda_free(cooVal)
          call cuda_free(xInd)
          call cuda_free(xVal)
          call cuda_free(y)
          call cuda_free(csrRowPtr)
          call cusparse_destroy_mat_descr(descrA)
          call cusparse_destroy(handle)
```

```
           write(*,*) "Conversion from COO to CSR format failed"
           stop
        endif
c       csrRowPtr = [0 3 4 7 9]

c       exercise Level 1 routines (scatter vector elements)
c       CUSPARSE_STATUS_SUCCESS=0
c       CUSPARSE_INDEX_BASE_ONE=1
        call get_shifted_address(y,n*8,ynp1)
        status= cusparse_dsctr(handle, nnz_vector, xVal, xInd,
      $                        ynp1, 1)
        if (status /= 0) then
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
           call cuda_free(xVal)
           call cuda_free(y)
           call cuda_free(csrRowPtr)
           call cusparse_destroy_mat_descr(descrA)
           call cusparse_destroy(handle)
           write(*,*) "Scatter from sparse to dense vector failed"
           stop
        endif
c       y = [10 20 30 40 | 100 200 70 400]

c       exercise Level 2 routines (csrmv)
c       CUSPARSE_STATUS_SUCCESS=0
c       CUSPARSE_OPERATION_NON_TRANSPOSE=0
        status= cusparse_dcsrmv(handle, 0, n, n, nnz, dtwo,
      $                        descrA, cooVal, csrRowPtr, cooColIndex,
      $                        y, dthree, ynp1)
        if (status /= 0) then
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
           call cuda_free(xVal)
           call cuda_free(y)
           call cuda_free(csrRowPtr)
           call cusparse_destroy_mat_descr(descrA)
           call cusparse_destroy(handle)
           write(*,*) "Matrix-vector multiplication failed"
           stop
        endif

c       print intermediate results (y)
c       y = [10 20 30 40 | 680 760 1230 2240]
c       cudaSuccess=0
c       cudaMemcpyDeviceToHost=2
        cudaStat1 = cuda_memcpy_c2fort_real(yHostPtr, y, 2*n*8, 2)
        if (cudaStat1 /= 0) then
           call cuda_free(cooRowIndex)
           call cuda_free(cooColIndex)
           call cuda_free(cooVal)
           call cuda_free(xInd)
           call cuda_free(xVal)
           call cuda_free(y)
           call cuda_free(csrRowPtr)
           call cusparse_destroy_mat_descr(descrA)
           call cusparse_destroy(handle)
           write(*,*) "Memcpy from Device to Host failed"
```

```
            stop
         endif
         write(*,*) "Intermediate results:"
         do j=1,2
            do i=1,n
                write(*,*) "yHostPtr[",i,",",j,"]=",yHostPtr(i+n*(j-1))
            enddo
         enddo

c        exercise Level 3 routines (csrmm)
c        cudaSuccess=0
c        CUSPARSE_STATUS_SUCCESS=0
c        CUSPARSE_OPERATION_NON_TRANSPOSE=0
         cudaStat1 = cuda_malloc(z, 2*(n+1)*8)
         if (cudaStat1 /= 0) then
            call cuda_free(cooRowIndex)
            call cuda_free(cooColIndex)
            call cuda_free(cooVal)
            call cuda_free(xInd)
            call cuda_free(xVal)
            call cuda_free(y)
            call cuda_free(csrRowPtr)
            call cusparse_destroy_mat_descr(descrA)
            call cusparse_destroy(handle)
            write(*,*) "Device malloc failed (z)"
            stop
         endif
         cudaStat1 = cuda_memset(z, 0, 2*(n+1)*8)
         if (cudaStat1 /= 0) then
            call cuda_free(cooRowIndex)
            call cuda_free(cooColIndex)
            call cuda_free(cooVal)
            call cuda_free(xInd)
            call cuda_free(xVal)
            call cuda_free(y)
            call cuda_free(z)
            call cuda_free(csrRowPtr)
            call cusparse_destroy_mat_descr(descrA)
            call cusparse_destroy(handle)
            write(*,*) "Memset on Device failed"
            stop
         endif
         status= cusparse_dcsrmm(handle, 0, n, 2, n, nnz, dfive,
     $                           descrA, cooVal, csrRowPtr, cooColIndex,
     $                           y, n, dzero, z, n+1)
         if (status /= 0) then
            call cuda_free(cooRowIndex)
            call cuda_free(cooColIndex)
            call cuda_free(cooVal)
            call cuda_free(xInd)
            call cuda_free(xVal)
            call cuda_free(y)
            call cuda_free(z)
            call cuda_free(csrRowPtr)
            call cusparse_destroy_mat_descr(descrA)
            call cusparse_destroy(handle)
            write(*,*) "Matrix-matrix multiplication failed"
            stop
         endif

c        print final results (z)
c        cudaSuccess=0
```

```
c      cudaMemcpyDeviceToHost=2
       cudaStat1 = cuda_memcpy_c2fort_real(zHostPtr, z, 2*(n+1)*8, 2)
       if (cudaStat1 /= 0) then
          call cuda_free(cooRowIndex)
          call cuda_free(cooColIndex)
          call cuda_free(cooVal)
          call cuda_free(xInd)
          call cuda_free(xVal)
          call cuda_free(y)
          call cuda_free(z)
          call cuda_free(csrRowPtr)
          call cusparse_destroy_mat_descr(descrA)
          call cusparse_destroy(handle)
          write(*,*) "Memcpy from Device to Host failed"
          stop
       endif
c      z = [950 400 2550 2600 0 | 49300 15200 132300 131200 0]
       write(*,*) "Final results:"
       do j=1,2
          do i=1,n+1
             write(*,*) "z[",i,",",j,"]=",zHostPtr(i+(n+1)*(j-1))
          enddo
       enddo

c      check the results
       epsilon = 0.00000000000001
       if ((DABS(zHostPtr(1) - 950.0)   .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(2) - 400.0)   .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(3) - 2550.0)  .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(4) - 2600.0)  .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(5) - 0.0)     .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(6) - 49300.0) .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(7) - 15200.0) .GT. epsilon)  .OR.
     $     (DABS(zHostPtr(8) - 132300.0).GT. epsilon)  .OR.
     $     (DABS(zHostPtr(9) - 131200.0).GT. epsilon)  .OR.
     $     (DABS(zHostPtr(10) - 0.0)    .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(1) - 10.0)    .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(2) - 20.0)    .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(3) - 30.0)    .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(4) - 40.0)    .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(5) - 680.0)   .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(6) - 760.0)   .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(7) - 1230.0)  .GT. epsilon)  .OR.
     $     (DABS(yHostPtr(8) - 2240.0)  .GT. epsilon)) then
           write(*,*) "fortran example test FAILED"
        else
           write(*,*) "fortran example test PASSED"
        endif

c       deallocate GPU memory and exit
        call cuda_free(cooRowIndex)
        call cuda_free(cooColIndex)
        call cuda_free(cooVal)
        call cuda_free(xInd)
        call cuda_free(xVal)
        call cuda_free(y)
        call cuda_free(z)
        call cuda_free(csrRowPtr)
        call cusparse_destroy_mat_descr(descrA)
        call cusparse_destroy(handle)

        stop
```

```
      end
```

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.