cuSPARSE Library
Table of Contents

Chapter 1. Introduction........................................................................................................ 1
  1.1. Naming Conventions.................................................................................................1
  1.2. Asynchronous Execution...........................................................................................2
  1.3. Static Library support...............................................................................................2

Chapter 2. Using the cuSPARSE API................................................................................... 3
  2.1. Thread Safety..............................................................................................................3
  2.2. Scalar Parameters.......................................................................................................3
  2.3. Parallelism with Streams...........................................................................................3
  2.4. Compatibility and Versioning.....................................................................................4
  2.5. Optimization Notes....................................................................................................4

Chapter 3. cuSPARSE Indexing and Data Formats............................................................. 5
  3.1. Index Base Format.................................................................................................... 5
    3.1.1. Vector Formats................................................................................................... 5
      3.1.1.1. Dense Format................................................................................................. 5
      3.1.1.2. Sparse Format............................................................................................... 5
    3.1.2. Matrix Formats.................................................................................................. 6
      3.2.1. Dense Format.................................................................................................... 6
      3.2.2. Coordinate Format (COO)................................................................................ 6
      3.2.3. Compressed Sparse Row Format (CSR).......................................................... 7
      3.2.4. Compressed Sparse Column Format (CSC)..................................................... 8
      3.2.5. Block Compressed Sparse Row Format (BSR)............................................... 9
      3.2.6. Extended BSR Format (BSRX)....................................................................... 11

Chapter 4. cuSPARSE Types Reference............................................................................ 13
  4.1. Data types................................................................................................................ 13
  4.2. cusparseStatus_t...................................................................................................... 13
  4.3. cusparseHandle_t...................................................................................................... 14
  4.4. cusparsePointerMode_t............................................................................................ 15
  4.5. cusparseOperation_t................................................................................................. 15
  4.6. cusparseAction_t....................................................................................................... 15
  4.7. cusparseDirection_t................................................................................................. 15
  4.8. cusparseMatDescr_t................................................................................................. 16
    4.8.1. cusparseDiagType_t......................................................................................... 16
    4.8.2. cusparseFillMode_t......................................................................................... 16
    4.8.3. cusparseIndexBase_t...................................................................................... 16
    4.8.4. cusparseMatrixType_t.................................................................................... 17
4.9. cusparseAlgMode_t................................................................................................................. 17
4.10. cusparseColorInfo_t............................................................................................................... 17
4.11. cusparseSolvePolicy_t............................................................................................................. 18
4.12. bsric02Info_t........................................................................................................................ 18
4.13. bsrilu02Info_t........................................................................................................................ 18
4.14. bsrsm2Info_t.......................................................................................................................... 18
4.15. bsrsv2Info_t........................................................................................................................... 18
4.16. csrgemm2Info_t.................................................................................................................... 18
4.17. csric02Info_t.......................................................................................................................... 18
4.18. csrilu02Info_t......................................................................................................................... 18
4.19. csrsm2Info_t.......................................................................................................................... 19
4.20. csrsv2Info_t............................................................................................................................ 19

Chapter 5. cuSPARSE Management Function Reference................................................. 20
5.1. cusparseCreate()..................................................................................................................... 20
5.2. cusparseDestroy().................................................................................................................... 20
5.3. cusparseGetErrorName()........................................................................................................ 20
5.4. cusparseGetErrorString()........................................................................................................ 21
5.5. cusparseGetProperty()............................................................................................................ 21
5.6. cusparseGetVersion().............................................................................................................. 21
5.7. cusparseGetPointerMode()..................................................................................................... 22
5.8. cusparseSetPointerMode()...................................................................................................... 22
5.9. cusparseGetStream().............................................................................................................. 22
5.10. cusparseSetStream()............................................................................................................. 23

Chapter 6. cuSPARSE Helper Function Reference............................................................ 24
6.1. cusparseCreateColorInfo()...................................................................................................... 24
6.2. cusparseCreateMatDescr()..................................................................................................... 24
6.3. cusparseDestroyColorInfo()................................................................................................... 24
6.4. cusparseDestroyMatDescr().................................................................................................. 25
6.5. cusparseGetMatDiagType()..................................................................................................... 25
6.6. cusparseGetMatFillMode()..................................................................................................... 25
6.7. cusparseGetMatIndexBase()................................................................................................... 26
6.8. cusparseGetMatType()............................................................................................................. 26
6.9. cusparseSetMatDiagType().................................................................................................... 26
6.10. cusparseSetMatFillMode().................................................................................................. 26
6.11. cusparseSetMatIndexBase()................................................................................................. 27
6.12. cusparseSetMatType().......................................................................................................... 27
6.13. cusparseCreateCsrsv2Info().................................................................................................. 27
6.14. cusparseDestroyCsrsv2Info()................................................................................................ 28
6.15. cusparseCreateCsrsm2Info() ............................................................... 28
6.16. cusparseDestroyCsrsm2Info() .......................................................... 28
6.17. cusparseCreateCsric02Info() .......................................................... 29
6.18. cusparseDestroyCsric02Info() ......................................................... 29
6.19. cusparseCreateCsrilu02Info() .......................................................... 29
6.20. cusparseDestroyCsrilu02Info() ......................................................... 29
6.21. cusparseCreateBsrsv2Info() .............................................................. 30
6.22. cusparseDestroyBsrsv2Info() ........................................................... 30
6.23. cusparseCreateBsrsm2Info() .............................................................. 30
6.24. cusparseDestroyBsrsm2Info() ........................................................... 30
6.25. cusparseCreateBsric02Info() ............................................................ 31
6.26. cusparseDestroyBsric02Info() .......................................................... 31
6.27. cusparseCreateBsrilu02Info() ............................................................ 31
6.28. cusparseDestroyBsrilu02Info() .......................................................... 31
6.29. cusparseCreateCsrgemm2Info() ...................................................... 32
6.30. cusparseDestroyCsrgemm2Info() ...................................................... 32
6.31. cusparseCreatePruneInfo() .............................................................. 33
6.32. cusparseDestroyPruneInfo() .............................................................. 33

Chapter 7. cuSPARSE Level 1 Function Reference ........................................ 34
7.1. cusparse<t>axpyi() [DEPRECATED] .................................................... 34
7.2. cusparse<t>gthr() [DEPRECATED] ....................................................... 35
7.3. cusparse<t>gthrz() [DEPRECATED] .................................................... 36
7.4. cusparse<t>roti() [DEPRECATED] ...................................................... 38
7.5. cusparse<t>sctr() [DEPRECATED] ...................................................... 39

Chapter 8. cuSPARSE Level 2 Function Reference ........................................ 41
8.1. cusparse<t>bsrmv() .............................................................................. 41
8.2. cusparse<t>bsrxmv() ........................................................................... 44
8.3. cusparse<t>bsrsv2_bufferSize() ....................................................... 48
8.4. cusparse<t>bsrsv2_analysis() ............................................................. 50
8.5. cusparse<t>bsrsv2_solve() ................................................................. 53
8.6. cusparseXbsrsv2_zeroPivot() ............................................................ 57
8.7. cusparseCsrsv2Ex() ............................................................................ 58
8.8. cusparse<t>csrsv2_bufferSize() ....................................................... 59
8.9. cusparse<t>csrsv2_analysis() ............................................................. 61
8.10. cusparse<t>csrsv2_solve() ............................................................... 64
8.11. cusparseXcsrsv2_zeroPivot() ........................................................... 67
8.12. cusparse<t>gemvi() ............................................................................ 68

Chapter 9. cuSPARSE Level 3 Function Reference ........................................ 72
14.1.2. cusparseFormat_t .........................................................................................................237
14.1.3. cusparseOrder_t ........................................................................................................... 237
14.1.4. cusparseIndexType_t .................................................................................................... 237

14.2. Sparse Vector APIs ............................................................................................................. 238
14.2.1. cusparseCreateSpVec().................................................................................................238
14.2.2. cusparseDestroySpVec()...............................................................................................238
14.2.3. cusparseSpVecGet()...................................................................................................... 239
14.2.4. cusparseSpVecGetIndexBase().....................................................................................239
14.2.5. cusparseSpVecGetValues()........................................................................................... 239
14.2.6. cusparseSpVecSetValues()........................................................................................... 240

14.3. Sparse Matrix APIs ..............................................................................................................240
14.3.1. cusparseCreateCoo().................................................................................................... 240
14.3.2. cusparseCreateCooAoS()..............................................................................................241
14.3.3. cusparseCreateCsr().....................................................................................................241
14.3.4. cusparseCreateCsc().................................................................................................... 242
14.3.5. cusparseDestroySpMat()............................................................................................ 243
14.3.6. cusparseCooGet()...........................................................................................................243
14.3.7. cusparseCooAosGet().....................................................................................................244
14.3.8. cusparseCsrGet()..........................................................................................................244
14.3.9. cusparseCsrSetPointers()............................................................................................ 245
14.3.10. cusparseSpMatGetSize()............................................................................................ 246
14.3.11. cusparseSpMatGetFormat()....................................................................................... 246
14.3.12. cusparseSpMatGetIndexBase().................................................................................. 246
14.3.13. cusparseSpMatGetValues().........................................................................................247
14.3.14. cusparseSpMatSetValue()..........................................................................................247
14.3.15. cusparseSpMatGetStridedBatch()..............................................................................247
14.3.16. cusparseSpMatSetStridedBatch() [DEPRECATED]..................................................... 248
14.3.17. cusparseCooSetStridedBatch().................................................................................. 248
14.3.18. cusparseCsrSetStridedBatch()...................................................................................248

14.4. Dense Vector APIs..............................................................................................................249
14.4.1. cusparseCreateDnVec().................................................................................................249
14.4.2. cusparseDestroyDnVec()..............................................................................................249
14.4.3. cusparseDnVecGet().....................................................................................................249
14.4.4. cusparseDnVecGetValues()..........................................................................................249
14.4.5. cusparseDnVecSetValues()..........................................................................................250

14.5. Dense Matrix APIs.............................................................................................................250
14.5.1. cusparseCreateDnMat().................................................................................................251
14.5.2. cusparseDestroyDnMat()..............................................................................................251
Chapter 1. Introduction

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. The library targets matrices with a number of (structural) zero elements which represent > 95% of the total entries.

It is implemented on top of the NVIDIA® CUDA™ runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++.

The library routines can be classified into four categories:

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

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

It is the responsibility of the developer to allocate memory and to copy data between GPU memory and CPU memory using standard CUDA runtime API routines, such as `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`, and `cudaMemcpyAsync()`.

1.1. Naming Conventions

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

```
cusparse<tt>[<matrix data format>]<operation>[<output matrix data format>]
```

where `<tt>` can be `S`, `D`, `C`, `Z`, or `X`, corresponding to the data types `float`, `double`, `cuComplex`, `cuDoubleComplex`, and the generic type, respectively.

The `<matrix data format>` can be `dense`, `coo`, `csr`, or `csc`, corresponding to the dense, coordinate, compressed sparse row, and compressed sparse column formats, respectively.
Finally, the operation can be axpyi, gthr, gthrz, roti, or sctr, corresponding to the Level 1 functions; it also can be mv or sv, corresponding to the Level 2 functions, as well as mm or sm, corresponding to the Level 3 functions.

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

1.2. Asynchronous Execution

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

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

1.3. Static Library support

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

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

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

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

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

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

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

Note that in the latter case, the library cuda is not needed. The CUDA Runtime will try to open explicitly the cuda library if needed. In the case of a system which does not have the CUDA driver installed, this allows the application to gracefully manage this issue and potentially run if a CPU-only path is available.
Chapter 2. Using the cuSPARSE API

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

2.1. Thread Safety

The library is thread safe and its functions can be called from multiple host threads. However, simultaneous read/writes of the same objects (or of the same handle) are not safe. Hence the handle must be private per thread, i.e., only one handle per thread is safe.

2.2. Scalar Parameters

In the cuSPARSE API, the scalar parameters $\alpha$ and $\beta$ can be passed by reference on the host or the device.

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

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

2.3. Parallelism with Streams

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

The application can conceptually associate a stream with each task. To achieve the overlap of computation between the tasks, the developer should create CUDA streams using the function $\text{cudaStreamCreate()}$ and set the stream to be used by each individual cuSPARSE library routine by calling $\text{cusparseSetStream()}$ just before calling the actual cuSPARSE routine. Then, computations performed in separate streams would be overlapped automatically on the
GPU, when possible. This approach is especially useful when the computation performed by a single task is relatively small and is not enough to fill the GPU with work, or when there is a data transfer that can be performed in parallel with the computation.

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

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

### 2.4. Compatibility and Versioning

The cuSPARSE APIs are intended to be backward compatible at the source level with future releases (unless stated otherwise in the release notes of a specific future release). In other words, if a program uses cuSPARSE, it should continue to compile and work correctly with newer versions of cuSPARSE without source code changes. cuSPARSE is not guaranteed to be backward compatible at the binary level. Using different versions of the cusparse.h header file and the shared library is not supported. Using different versions of cuSPARSE and the CUDA runtime is not supported. The APIs should be backward compatible at the source level for public functions in most cases.

### 2.5. Optimization Notes

Most of the cuSPARSE routines can be optimized by exploiting **CUDA Graphs capture** and **Hardware Memory Compression** features.

More in details, a single cuSPARSE call or a sequence of calls can be captured by a CUDA Graph and executed in a second moment. This minimizes kernels launch overhead and allows the CUDA runtime to optimize the whole workflow. A full example of CUDA graphs capture applied to a cuSPARSE routine can be found in cuSPARSE Library Samples - CUDA Graph.

Secondly, the data types and functionalities involved in cuSPARSE are suitable for **Hardware Memory Compression** available in Ampere GPU devices (compute capability 8.0) or above. The feature allows memory compression for data with enough zero bytes without no loss of information. The device memory must be allocation with the CUDA driver APIs. A full example of Hardware Memory Compression applied to a cuSPARSE routine can be found in cuSPARSE Library Samples - Memory Compression.
Chapter 3. cuSPARSE Indexing and Data Formats

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

3.1. Index Base Format

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

3.1.1. Vector Formats

This section describes dense and sparse vector formats.

3.1.1.1. Dense Format

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

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

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

3.1.1.2. Sparse Format

Sparse vectors are represented with two arrays.

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

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

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

$$[1 \ 4 \ 5 \ 7 ]$$
It can also be stored as a sparse vector with zero-based indexing.

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

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

### 3.2. Matrix Formats

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

#### 3.2.1. Dense Format

The dense matrix \( X \) is assumed to be stored in column-major format in memory and is represented by the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>integer</td>
</tr>
<tr>
<td>( n )</td>
<td>integer</td>
</tr>
<tr>
<td>( ldX )</td>
<td>integer</td>
</tr>
<tr>
<td>( X )</td>
<td>pointer</td>
</tr>
</tbody>
</table>

For example, \( m \times n \) dense matrix \( X \) with leading dimension \( ldX \) can be stored with one-based indexing as shown.

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

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

\[
\begin{bmatrix}
X_{1,1} & X_{2,1} & \cdots & X_{m,1} & X_{ldX,1} & \cdots & X_{1,n} & X_{2,n} & \cdots & X_{m,n} & X_{ldX,n}
\end{bmatrix}
\]

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

#### 3.2.2. Coordinate Format (COO)

The \( m \times n \) sparse matrix \( A \) is represented in COO format by the following parameters.
cuSPARSE Indexing and Data Formats

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nnz</code></td>
<td>integer</td>
<td>The number of nonzero elements in the matrix.</td>
</tr>
<tr>
<td><code>cooValA</code></td>
<td>pointer</td>
<td>Points to the data array of length <code>nnz</code> that holds all nonzero values of <code>A</code> in row-major format.</td>
</tr>
<tr>
<td><code>cooRowIndA</code></td>
<td>pointer</td>
<td>Points to the integer array of length <code>nnz</code> that contains the row indices of the corresponding elements in array <code>cooValA</code>.</td>
</tr>
<tr>
<td><code>cooColIndA</code></td>
<td>pointer</td>
<td>Points to the integer array of length <code>nnz</code> that contains the column indices of the corresponding elements in array <code>cooValA</code>.</td>
</tr>
</tbody>
</table>

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

For example, consider the following $4 \times 5$ matrix $A$.

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

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

- `cooValA` = $[1.0 \ 4.0 \ 2.0 \ 3.0 \ 5.0 \ 7.0 \ 8.0 \ 9.0 \ 6.0]$ 
- `cooRowIndA` = $[0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 3 \ 3]$ 
- `cooColIndA` = $[0 \ 1 \ 1 \ 2 \ 0 \ 3 \ 4 \ 2 \ 4]$ 

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

- `cooValA` = $[1.0 \ 4.0 \ 2.0 \ 3.0 \ 5.0 \ 7.0 \ 8.0 \ 9.0 \ 6.0]$ 
- `cooRowIndA` = $[1 \ 1 \ 2 \ 2 \ 3 \ 3 \ 4 \ 4]$ 
- `cooColIndA` = $[1 \ 2 \ 2 \ 3 \ 1 \ 4 \ 5 \ 3 \ 5]$ 

### 3.2.3. Compressed Sparse Row Format (CSR)

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nnz</code></td>
<td>integer</td>
<td>The number of nonzero elements in the matrix.</td>
</tr>
<tr>
<td><code>csrValA</code></td>
<td>pointer</td>
<td>Points to the data array of length <code>nnz</code> that holds all nonzero values of <code>A</code> in row-major format.</td>
</tr>
<tr>
<td><code>csrRowPtrA</code></td>
<td>pointer</td>
<td>Points to the integer array of length $m+1$ that holds indices into the arrays <code>csrColIndA</code> and <code>csrValA</code>. The first $m$ entries of this array contain the indices of the first nonzero element in the $i$th row for $i=0,...,m$, while the last entry contains $nnz+csrRowPtrA(0)$. In general, <code>csrRowPtrA(0)</code> is 0 or 1 for zero- and one-based indexing, respectively.</td>
</tr>
<tr>
<td><code>csrColIndA</code></td>
<td>pointer</td>
<td>Points to the integer array of length <code>nnz</code> that contains the column indices of the corresponding elements in array <code>csrValA</code>.</td>
</tr>
</tbody>
</table>
Sparse matrices in CSR format are assumed to be stored in row-major CSR format, in other words, the index arrays are first sorted by row indices and then within the same row by column indices. It is assumed that each pair of row and column indices appears only once.

Consider again the $4 \times 5$ matrix $A$.

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

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

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

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

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

### 3.2.4. Compressed Sparse Column Format (CSC)

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{nnz}$</td>
<td>integer</td>
<td>The number of nonzero elements in the matrix.</td>
</tr>
<tr>
<td>$\text{cscValA}$</td>
<td>pointer</td>
<td>Points to the data array of length $\text{nnz}$ that holds all nonzero values of $A$ in column-major format.</td>
</tr>
<tr>
<td>$\text{cscRowIndA}$</td>
<td>pointer</td>
<td>Points to the integer array of length $\text{nnz}$ that contains the row indices of the corresponding elements in array $\text{cscValA}$.</td>
</tr>
<tr>
<td>$\text{cscColPtrA}$</td>
<td>pointer</td>
<td>Points to the integer array of length $n+1$ that holds indices into the arrays $\text{cscRowIndA}$ and $\text{cscValA}$. The first $n$ entries of this array contain the indices of the first nonzero element in the $i$th row for $i=1,\ldots,n$, while the last entry contains $\text{nnz}+\text{cscColPtrA}(0)$. In general, $\text{cscColPtrA}(0)$ is 0 or 1 for zero- and one-based indexing, respectively.</td>
</tr>
</tbody>
</table>

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 $4 \times 5$ matrix $A$.

$$
\begin{bmatrix}
1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\
5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\
0.0 & 0.0 & 9.0 & 0.0 & 6.0
\end{bmatrix}
$$
It is stored in CSC format with zero-based indexing this way.

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

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

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

Each pair of row and column indices appears only once.

### 3.2.5. Block Compressed Sparse Row Format (BSR)

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

\(A\) is represented in BSR format by the following parameters.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockDim</td>
<td>integer</td>
<td>Block dimension of matrix (A).</td>
</tr>
<tr>
<td>mb</td>
<td>integer</td>
<td>The number of block rows of (A).</td>
</tr>
<tr>
<td>nb</td>
<td>integer</td>
<td>The number of block columns of (A).</td>
</tr>
<tr>
<td>nnzb</td>
<td>integer</td>
<td>The number of nonzero blocks in the matrix.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>pointer</td>
<td>Points to the data array of length (nnzb \times \text{blockDim}^2) that holds all elements of nonzero blocks of (A). The block elements are stored in either column-major order or row-major order.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>pointer</td>
<td>Points to the integer array of length (mb+1) that holds indices into the arrays bsrColIndA and bsrValA. The first (mb) entries of this array contain the indices of the first nonzero block in the (i)th block row for (i=1,\ldots,mb), while the last entry contains (nnzb+\text{bsrRowPtrA}(0)). In general, (\text{bsrRowPtrA}(0)) is 0 or 1 for zero- and one-based indexing, respectively.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>pointer</td>
<td>Points to the integer array of length (nnzb) that contains the column indices of the corresponding blocks in array bsrValA.</td>
</tr>
</tbody>
</table>

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

For example, consider again the \(4 \times 5\) matrix \(A\).

\[
\begin{bmatrix}
1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\
5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\
0.0 & 0.0 & 9.0 & 0.0 & 6.0
\end{bmatrix}
\]
If \( blockDim \) is equal to 2, then \( mb \) is 2, \( nb \) is 3, and matrix \( A \) is split into \( 2 \times 3 \) block matrix \( A_b \). The dimension of \( A_b \) is \( 4 \times 6 \), slightly bigger than matrix \( A \), so zeros are filled in the last column of \( A_b \). The element-wise view of \( A_b \) is this.

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

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

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

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

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

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

- \( \text{bsrValA} = [A_{00}, A_{01}, A_{10}, A_{11}, A_{12}] \)
- \( \text{bsrRowPtrA} = [0, 2, 5] \)
- \( \text{bsrColIndA} = [0, 1, 0, 1, 2] \)

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

\( \text{bsrValA} = [1 \ 0 \ 4 \ 2 \ 0 \ 3 \ 0 \ 0 \ 5 \ 0 \ 0 \ 0 \ 0 \ 9 \ 7 \ 0 \ 8 \ 6 \ 0 \ 0 \ 0] \)

Under row-major order, the physical storage of \( \text{bsrValA} \) is this.

\( \text{bsrValA} = [1 \ 4 \ 0 \ 2 \ 0 \ 0 \ 3 \ 0 \ 5 \ 0 \ 0 \ 0 \ 0 \ 7 \ 9 \ 0 \ 8 \ 0 \ 6 \ 0 \ 0] \)

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

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

\[
\text{bsrValA} = [1 \ 0 \ 4 \ 2 \ 0 \ 3 \ 0 \ 0 \ 5 \ 0 \ 0 \ 0 \ 0 \ 9 \ 7 \ 0 \ 8 \ 6 \ 0 \ 0 \ 0]
\]

- \( \text{bsrRowPtrA} = [1 \ 3 \ 6] \)
- \( \text{bsrColIndA} = [1 \ 2 \ 1 \ 2 \ 3] \)

**Note:** The general BSR format has two parameters, \( \text{rowBlockDim} \) and \( \text{colBlockDim} \). \( \text{rowBlockDim} \) is number of rows within a block and \( \text{colBlockDim} \) is number of columns within a block. If \( \text{rowBlockDim} = \text{colBlockDim} \), general BSR format is the same as BSR format. If
3.2.6. Extended BSR Format (BSRX)

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>blockDim</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>mb</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>nb</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>nnzb</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>bsrValA</code></td>
<td>pointer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bsrRowPtrA</code></td>
<td>pointer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bsrEndPtrA</code></td>
<td>pointer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bsrColIndA</code></td>
<td>pointer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

Assume it has this BSR format.

$$ \begin{align*}
  \text{bsrValA of BSR} &= [A_{00} \ A_{01} \ A_{10} \ A_{11} \ A_{12}] \\
  \text{bsrRowPtrA of BSR} &= [0 \ 2 \ 5] \\
  \text{bsrColIndA of BSR} &= [0 \ 1 \ 0 \ 1 \ 2]
\end{align*} $$
The `bsrRowPtrA` of the BSRX format is simply the first two elements of the `bsrRowPtrA` BSR format. The `bsrEndPtrA` of BSRX format is the last two elements of the `bsrRowPtrA` of BSR format.

\[
\text{bsrRowPtrA of BSRX} = [0 \quad 2] \\
\text{bsrEndPtrA of BSRX} = [2 \quad 5]
\]

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

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

\[
\text{bsrValA of } \hat{A} = [A_{00} \quad A_{01} \quad A_{10} \quad A_{11} \quad A_{12}] \\
\text{bsrColIndA of } \hat{A} = [0 \quad 1 \quad 0 \quad 1 \quad 2] \\
\text{bsrRowPtrA of } \hat{A} = [0 \quad 3] \\
\text{bsrEndPtrA of } \hat{A} = [0 \quad 4]
\]
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. cusparseStatus_t

This data type represents the status returned by the library functions and it can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_STATUS_SUCCESS</td>
<td>The operation completed successfully</td>
</tr>
<tr>
<td></td>
<td>The cuSPARSE library was not initialized. This is usually caused by the lack of a prior call, an error in the CUDA Runtime API called by the cuSPARSE routine, or an error in the hardware setup.</td>
</tr>
<tr>
<td></td>
<td><strong>To correct:</strong> 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.</td>
</tr>
<tr>
<td></td>
<td>The error also applies to generic APIs [Generic APIs reference] for indicating a matrix/vector descriptor not initialized.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_NOT_INITIALIZED</td>
<td>Resource allocation failed inside the cuSPARSE library. This is usually caused by a device memory allocation [cudaMalloc()] or by a host memory allocation failure.</td>
</tr>
<tr>
<td></td>
<td><strong>To correct:</strong> prior to the function call, deallocate previously allocated memory as much as possible.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_ALLOC_FAILED</td>
<td>An unsupported value or parameter was passed to the function [a negative vector size, for example]</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_ARCH_MISMATCH</td>
<td>The function requires a feature absent from the device architecture. To correct: compile and run the application on a device with appropriate compute capability.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_EXECUTION_FAILED</td>
<td>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.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_INTERNAL_ERROR</td>
<td>An internal cuSPARSE operation failed. 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 completion.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>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.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_NOT_SUPPORTED</td>
<td>The operation or data type combination is currently not supported by the function.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_INSUFFICIENT_RESOURCES</td>
<td>The resources for the computation, such as GPU global or shared memory, are not sufficient to complete the operation. The error can also indicate that the current computation mode [e.g. bit size of sparse matrix indices] does not allow to handle the given input.</td>
</tr>
</tbody>
</table>

4.3. **cusparseHandle_t**

This is a pointer type to an opaque cuSPARSE context, which the user must initialize by calling prior to calling `cusparseCreate()` any other library function. The handle created and returned by `cusparseCreate()` must be passed to every cuSPARSE function.
4.4. **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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_POINTER_MODE_HOST</td>
<td>the scalars are passed by reference on the host.</td>
</tr>
<tr>
<td>CUSPARSE_POINTER_MODE_DEVICE</td>
<td>the scalars are passed by reference on the device.</td>
</tr>
</tbody>
</table>

4.5. **cusparseOperation_t**

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_OPERATION_NON_TRANSPOSE</td>
<td>the non-transpose operation is selected.</td>
</tr>
<tr>
<td>CUSPARSE_OPERATION_TRANSPOSE</td>
<td>the transpose operation is selected.</td>
</tr>
<tr>
<td>CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE</td>
<td>the conjugate transpose operation is selected.</td>
</tr>
</tbody>
</table>

4.6. **cusparseAction_t**

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_ACTION_SYMBOLIC</td>
<td>the operation is performed only on indices.</td>
</tr>
<tr>
<td>CUSPARSE_ACTION_NUMERIC</td>
<td>the operation is performed on data and indices.</td>
</tr>
</tbody>
</table>

4.7. **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[SIDICIZ]nnz`. Besides storage format of blocks in BSR format is also controlled by this type.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_DIRECTION_ROW</td>
<td>the matrix should be parsed by rows.</td>
</tr>
</tbody>
</table>
### Value | Meaning
---|---
CUSPARSE_DIRECTION_COLUMN | the matrix should be parsed by columns.

## 4.8. cusparseMatDescr_t

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

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

### 4.8.1. cusparseDiagType_t

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_DIAG_TYPE_NON_UNIT</td>
<td>the matrix diagonal has non-unit elements.</td>
</tr>
<tr>
<td>CUSPARSE_DIAG_TYPE_UNIT</td>
<td>the matrix diagonal has unit elements.</td>
</tr>
</tbody>
</table>

### 4.8.2. cusparseFillMode_t

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_FILL_MODE_LOWER</td>
<td>the lower triangular part is stored.</td>
</tr>
<tr>
<td>CUSPARSE_FILL_MODE_UPPER</td>
<td>the upper triangular part is stored.</td>
</tr>
</tbody>
</table>

### 4.8.3. cusparseIndexBase_t

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_INDEX_BASE_ZERO</td>
<td>the base index is zero.</td>
</tr>
<tr>
<td>CUSPARSE_INDEX_BASE_ONE</td>
<td>the base index is one.</td>
</tr>
</tbody>
</table>
4.8.4. **cusparseMatrixType_t**

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

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

In general, SpMV, preconditioners (incomplete Cholesky or incomplete LU) and triangular solver are combined together in iterative solvers, for example PCG and GMRES. If the user always uses general matrix (instead of symmetric matrix), there is no need to support other than general matrix in preconditioners. Therefore the new routines, [bsr|csr]sv2 (triangular solver), [bsr|csr]ilu02 (incomplete LU) and [bsr|csr]ic02 (incomplete Cholesky), only support matrix type CUSPARSE_MATRIX_TYPE_GENERAL.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_MATRIX_TYPE_GENERAL</td>
<td>the matrix is general.</td>
</tr>
<tr>
<td>CUSPARSE_MATRIX_TYPE_SYMMETRIC</td>
<td>the matrix is symmetric.</td>
</tr>
<tr>
<td>CUSPARSE_MATRIX_TYPE_HERMITIAN</td>
<td>the matrix is Hermitian.</td>
</tr>
<tr>
<td>CUSPARSE_MATRIX_TYPE_TRIANGULAR</td>
<td>the matrix is triangular.</td>
</tr>
</tbody>
</table>

4.9. **cusparseAlgMode_t**

This is type for algorithm parameter to cusparseCsrmvEx() and cusparseCsrmvEx_bufferSize() functions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_ALG_MERGE_PATH</td>
<td>Use load-balancing algorithm that suits better for irregular nonzero-patterns.</td>
</tr>
</tbody>
</table>

4.10. **cusparseColorInfo_t**

This is a pointer type to an opaque structure holding the information used in csrcolor().
4.11. cusparseSolvePolicy_t

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SOLVE_POLICY_NO_LEVEL</td>
<td>no level information is generated and used.</td>
</tr>
<tr>
<td>CUSPARSE_SOLVE_POLICY_USE_LEVEL</td>
<td>generate and use level information.</td>
</tr>
</tbody>
</table>

4.12. bsric02Info_t

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

4.13. bsrilu02Info_t

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

4.14. bsrsm2Info_t

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

4.15. bsrsv2Info_t

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

4.16. csrgemm2Info_t

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

4.17. csric02Info_t

This is a pointer type to an opaque structure holding the information used in csric02_bufferSize(), csric02_analysis(), and csric02().
4.18. **csrilu02Info_t**

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

4.19. **csrsm2Info_t**

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

4.20. **csrsrsv2Info_t**

This is a pointer type to an opaque structure holding the information used in csrsrsv2_bufferSize(), csrsrsv2_analysis(), and csrsrsv2_solve().
Chapter 5. cuSPARSE Management

Function Reference

The cuSPARSE functions for managing the library are described in this section.

5.1. cusparseCreate()

cusparseStatus_t
CUSPARSE_API 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.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The pointer to the handle to the cuSPARSE context</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status.

5.2. cusparseDestroy()

cusparseStatus_t
CUSPARSE_API 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.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The handle to the cuSPARSE context</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status.

5.3. cusparseGetErrorName()

const char*
cusparseGetErrorString(cusparseStatus_t status)

The function returns the string representation of an error code enum name. If the error code is not recognized, “unrecognized error code” is returned.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>IN</td>
<td>Error code to convert to string</td>
</tr>
<tr>
<td>const char*</td>
<td>OUT</td>
<td>Pointer to a NULL-terminated string</td>
</tr>
</tbody>
</table>

5.4. cusparseGetErrorString()

const char*
cusparseGetErrorString(cusparseStatus_t status)

Returns the description string for an error code. If the error code is not recognized, “unrecognized error code” is returned.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>IN</td>
<td>Error code to convert to string</td>
</tr>
<tr>
<td>const char*</td>
<td>OUT</td>
<td>Pointer to a NULL-terminated string</td>
</tr>
</tbody>
</table>

5.5. cusparseGetProperty()

cusparseStatus_t
cusparseGetProperty(libraryPropertyType type, int* value)

The function returns the value of the requested property. Refer to libraryPropertyType for supported types.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>IN</td>
<td>Requested property</td>
</tr>
<tr>
<td>value</td>
<td>OUT</td>
<td>Value of the requested property</td>
</tr>
</tbody>
</table>

libraryPropertyType (defined in library_types.h):

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAJOR_VERSION</td>
<td>Enumerator to query the major version</td>
</tr>
<tr>
<td>MINOR_VERSION</td>
<td>Enumerator to query the minor version</td>
</tr>
<tr>
<td>PATCH_LEVEL</td>
<td>Number to identify the patch level</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

5.6. cusparseGetVersion()
5.7. 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.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The handle to the cuSPARSE context</td>
</tr>
<tr>
<td>mode</td>
<td>OUT</td>
<td>One of the enumerated pointer mode types</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

5.8. 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.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The handle to the cuSPARSE context</td>
</tr>
<tr>
<td>mode</td>
<td>IN</td>
<td>One of the enumerated pointer mode types</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

5.9. cusparseGetStream()

cusparseStatus_t
cusparseGetStream(cusparseHandle_t handle, cudaStream_t *streamId)
This function gets the cuSPARSE library stream, which is being used to execute all calls to the cuSPARSE library functions. If the cuSPARSE library stream is not set, all kernels use the default NULL stream.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The handle to the cuSPARSE context</td>
</tr>
<tr>
<td>streamId</td>
<td>OUT</td>
<td>The stream used by the library</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

5.10. cusparseSetStream()

```c
 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.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>The handle to the cuSPARSE context</td>
</tr>
<tr>
<td>streamId</td>
<td>IN</td>
<td>The stream to be used by the library</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status
Chapter 6. cuSPARSE Helper Function Reference

The cuSPARSE helper functions are described in this section.

6.1. cusparseCreateColorInfo()

This function creates and initializes the cusparseColorInfo_t structure to default values.

```
cusparseStatus_t
cusparseCreateColorInfo(cusparseColorInfo_t* info)
```

Input

| info | the pointer to the cusparseColorInfo_t structure |

See cusparseStatus_t for the description of the return status.

6.2. cusparseCreateMatDescr()

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.

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

Input

| descrA | the pointer to the matrix descriptor. |

See cusparseStatus_t for the description of the return status.

6.3. cusparseDestroyColorInfo()

```
cusparseStatus_t
cusparseDestroyColorInfo(cusparseColorInfo_t info)
```
This function destroys and releases any memory required by the structure.

**Input**

| info | the pointer to the structure of csrColor() |

See [cusparseStatus_t](#) for the description of the return status

### 6.4. cusparseDestroyMatDescr()

cusparseStatus_t
cusparseDestroyMatDescr(cusparseMatDescr_t descrA)

This function releases the memory allocated for the matrix descriptor.

**Input**

| descrA | the matrix descriptor. |

See [cusparseStatus_t](#) for the description of the return status

### 6.5. 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.

### 6.6. 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.
6.7. 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. |

6.8. 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. |

6.9. 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. |

See [cusparseStatus_t](#) for the description of the return status.

6.10. cusparseSetMatFillMode()

cusparseStatus_t

cusparseSetMatFillMode(cusparseMatDescr_t descrA,
**6.11. 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.

See [cusparseStatus_t](#) for the description of the return status

**6.12. 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.

See [cusparseStatus_t](#) for the description of the return status

**6.13. cusparseCreateCsrsv2Info()**

cusparseStatus_t
cusparseCreateCsrsv2Info(csrsv2Info_t *info)

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.14. cusparseDestroyCsrsv2Info()

cusparseStatus_t
cusparseDestroyCsrsv2Info(csrsv2Info_t info);

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.15. cusparseCreateCsrsm2Info()

cusparseStatus_t
cusparseCreateCsrsm2Info(csrsm2Info_t *info);

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.16. cusparseDestroyCsrsm2Info()

cusparseStatus_t
cusparseDestroyCsrsm2Info(csrsm2Info_t info);

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

**Input**

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

See `cusparseStatus_t` for the description of the return status
6.17. cusparseCreateCsric02Info()

cusparseStatus_t
cusparseCreateCsric02Info(csric02Info_t *info);

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

**Input**

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

See cusparseStatus_t for the description of the return status

6.18. cusparseDestroyCsric02Info()

cusparseStatus_t
cusparseDestroyCsric02Info(csric02Info_t info);

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

**Input**

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

See cusparseStatus_t for the description of the return status

6.19. cusparseCreateCsrilu02Info()

cusparseStatus_t
cusparseCreateCsrilu02Info(csrilu02Info_t *info);

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

**Input**

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

See cusparseStatus_t for the description of the return status

6.20. cusparseDestroyCsrilu02Info()
cusparseDestroyCsrilu02Info(csrilu02Info_t info);

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.21. cusparseCreateBsrsv2Info()

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

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.22. cusparseDestroyBsrsv2Info()

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

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

**Input**

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

See `cusparseStatus_t` for the description of the return status

### 6.23. cusparseCreateBsrsm2Info()

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

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

**Input**

| info         | the pointer to the solve and analysis structure of bsrsm2. |
6.24. **cusparseDestroyBsrsm2Info()**

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

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

**Input**

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

See **cusparseStatus_t** for the description of the return status.

6.25. **cusparseCreateBsric02Info()**

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

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

**Input**

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

See **cusparseStatus_t** for the description of the return status.

6.26. **cusparseDestroyBsric02Info()**

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

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

**Input**

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

See **cusparseStatus_t** for the description of the return status.

6.27. **cusparseCreateBsrilu02Info()**

```c
 cusparseStatus_t
```

See **cusparseStatus_t** for the description of the return status.
cusparseCreateBsrilu02Info(bsrilu02Info_t *info);

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

**Input**

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

See [cusparseStatus_t](#) for the description of the return status

### 6.28. cusparseDestroyBsrilu02Info()

cusparseStatus_t
cusparseDestroyBsrilu02Info(bsrilu02Info_t info);

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

**Input**

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

See [cusparseStatus_t](#) for the description of the return status

### 6.29. cusparseCreateCsrgemm2Info()

cusparseStatus_t
cusparseCreateCsrgemm2Info(csrgemm2Info_t *info);

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

**Input**

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

See [cusparseStatus_t](#) for the description of the return status

### 6.30. cusparseDestroyCsrgemm2Info()

cusparseStatus_t
cusparseDestroyCsrgemm2Info(csrgemm2Info_t info);

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

**Input**

| info       | opaque structure of csrgemm2. |
6.31. cusparseCreatePruneInfo()

cusparseStatus_t cusparseCreatePruneInfo(pruneInfo_t *info);

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

Input

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

See cusparseStatus_t for the description of the return status

6.32. cusparseDestroyPruneInfo()

cusparseStatus_t cusparseDestroyPruneInfo(pruneInfo_t info);

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

Input

| info       | the structure of prune. |

See cusparseStatus_t for the description of the return status
Chapter 7. cuSPARSE Level 1 Function Reference

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

### 7.1. cusparse<t>axpyi() [DEPRECATED]

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

```c

usparseStatus_t cusparseSaxpyi(cusparseHandle_t handle, int nnz, const float* alpha, const float* xVal, const int* xInd, float* y, cusparseIndexBase_t idxBase)

usparseStatus_t cusparseDaxpyi(cusparseHandle_t handle, int nnz, const double* alpha, const double* xVal, const int* xInd, double* y, cusparseIndexBase_t idxBase)

usparseStatus_t cusparseCaxpyi(cusparseHandle_t handle, int nnz, const cuComplex* alpha, const cuComplex* xVal, const int* xInd, cuComplex* y, cusparseIndexBase_t idxBase)

usparseStatus_t cusparseZaxpyi(cusparseHandle_t handle, int nnz, const cuDoubleComplex* alpha, const cuDoubleComplex* xVal,
```
const int* xInd,
cuDenseComplex* y,
cusparseIndexBase_t idxBase)

This function multiplies the vector \( x \) in sparse format by the constant \( \alpha \) and adds the result to the vector \( y \) in dense format. This operation can be written as

\[
y = y + \alpha \times x
\]

In other words,

\[
\text{for } i=0 \text{ to } \text{nnz-1} \\
y[xInd[i]-idxBase] = y[xInd[i]-idxBase] + \alpha \times xVal[i]
\]

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of elements in vector ( x ).</td>
</tr>
<tr>
<td>alpha</td>
<td>(&lt;\text{type}&gt;) scalar used for multiplication.</td>
</tr>
<tr>
<td>xVal</td>
<td>(&lt;\text{type}&gt;) vector with ( \text{nnz} ) nonzero values of vector ( x ).</td>
</tr>
<tr>
<td>xInd</td>
<td>integer vector with ( \text{nnz} ) indices of the nonzero values of vector ( x ).</td>
</tr>
<tr>
<td>y</td>
<td>(&lt;\text{type}&gt;) vector in dense format.</td>
</tr>
<tr>
<td>idxBase</td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>(&lt;\text{type}&gt;) updated vector in dense format (that is unchanged if ( \text{nnz} == 0 )).</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.

### 7.2. cusparse<t>gthr() [DEPRECATED]

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

```c
cusparseStatus_t cusparseGthr(cusparseHandle_t handle, int nnz, const float* y, float* xVal,
const int* xInd, cusparseIndexBase_t idxBase)
```

```c
cusparseStatus_t cusparseGthr(cusparseHandle_t handle, int nnz, const int* xInd, cusparseIndexBase_t idxBase)
```
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 \( x\text{Ind} \) into the data array \( x\text{Val} \).

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of elements in vector ( x ).</td>
</tr>
<tr>
<td>( y )</td>
<td>(&lt;\text{type}&gt; ) vector in dense format [of size( \geq \max(x\text{Ind})-idxBase+1 )].</td>
</tr>
<tr>
<td>( x\text{Ind} )</td>
<td>integer vector with ( nnz ) indices of the nonzero values of vector ( x ).</td>
</tr>
<tr>
<td>idxBase</td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x\text{Val} )</td>
<td>(&lt;\text{type}&gt; ) vector with ( nnz ) nonzero values that were gathered from vector ( y ) [that is unchanged if ( nnz == 0 )].</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 7.3. cusparse<t>gthrz() [DEPRECATED]

[[DEPRECATED]] use [cusparseGather()](#) instead. *The routine will be removed in the next major release*
This function gathers the elements of the vector `y` listed in the index array `xInd` into the data array `xVal`. Also, it zeros out the gathered elements in the vector `y`.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>number of elements in vector <code>x</code>.</td>
</tr>
<tr>
<td><code>y</code></td>
<td><code>&lt;type&gt;</code> vector in dense format [of size(\leq\max(xInd))-idxBase+1].</td>
</tr>
<tr>
<td><code>xInd</code></td>
<td>integer vector with <code>nnz</code> indices of the nonzero values of vector <code>x</code>.</td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td><code>CUSPARSE_INDEX_BASE_ZERO</code> or <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xVal</code></td>
<td><code>&lt;type&gt;</code> vector with <code>nnz</code> nonzero values that were gathered from vector <code>y</code> [that is unchanged if <code>nnz == 0</code>].</td>
</tr>
</tbody>
</table>
7.4. **cusparse<t>roti() [DEPRECATED]**

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

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

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

This function applies the Givens rotation matrix

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

...to sparse \(x\) and dense \(y\) vectors. In other words,

\[
\begin{align*}
y[\text{Ind}[i]-\text{idxBase}] &= c \cdot y[\text{Ind}[i]-\text{idxBase}] - s \cdot x[i] \\
x[i] &= c \cdot x[i] + s \cdot y[\text{Ind}[i]-\text{idxBase}]
\end{align*}
\]

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of elements in vector (x).</td>
</tr>
<tr>
<td>xVal</td>
<td>(&lt;\text{type}&gt;) vector with (nnz) nonzero values of vector (x).</td>
</tr>
<tr>
<td>xInd</td>
<td>integer vector with (nnz) indices of the nonzero values of vector (x).</td>
</tr>
<tr>
<td>y</td>
<td>(&lt;\text{type}&gt;) vector in dense format.</td>
</tr>
</tbody>
</table>
c | cosine element of the rotation matrix.
---|---
s | sine element of the rotation matrix.
idxBase | CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.

### Output

|     |  
|-----|---------------------------|
| xVal | <type> updated vector in sparse format [that is unchanged if nnz == 0]. |
| y   | <type> updated vector in dense format [that is unchanged if nnz == 0]. |

See `cusparseStatus_t` for the description of the return status.

### 7.5. cusparse<t>sctr() [DEPRECATED]

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

```c
 cusparseStatus_t cusparseSsctr(cusparseHandle_t    handle,
 int                 nnz,
 const float*        xVal,
 const int*          xInd,
 float*              y,
 cusparseIndexBase_t idxBase)
```

```c
 cusparseStatus_t cusparseDsctr(cusparseHandle_t    handle,
 int                 nnz,
 const double*       xVal,
 const int*          xInd,
 double*             y,
 cusparseIndexBase_t idxBase)
```

```c
 cusparseStatus_t cusparseCsctr(cusparseHandle_t    handle,
 int                 nnz,
 const cuComplex*    xVal,
 const int*          xInd,
 cuComplex*          y,
 cusparseIndexBase_t idxBase)
```

```c
 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 \( xInd \).

- The routine requires no extra storage.
The routine supports asynchronous execution
The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of elements in vector x.</td>
</tr>
<tr>
<td>xVal</td>
<td>&lt;type&gt; vector with nnz nonzero values of vector x.</td>
</tr>
<tr>
<td>xInd</td>
<td>integer vector with nnz indices of the nonzero values of vector x.</td>
</tr>
<tr>
<td>y</td>
<td>&lt;type&gt; dense vector [of size ≥ max(xInd) - idxBase + 1].</td>
</tr>
<tr>
<td>idxBase</td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>&lt;type&gt; vector with nnz nonzero values that were scattered from vector x [that is unchanged if nnz == 0].</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.
Chapter 8. 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 `csrsv2_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 `csrsv2Info_t` that has been initialized previously with a call to `cusparseCreateCsrsv2Info()`.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the `csrsv2Info_t` parameter by calling the appropriate `csrsv2_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 `csrsv2Info_t` parameter can be released by calling `cusparseDestroyCsrsv2Info()`.

8.1. `cusparse<t>bsrmv()`

```c
CUSPARSE_STATUS_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,
```
This function performs the matrix-vector operation

\[ y = \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y \]

where \( A \) is an \((mb \times blockDim) \times (nb \times blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays \( \text{bsrVal} \), \( \text{bsrRowPtr} \), and \( \text{bsrColInd} \); \( x \) and \( y \) are vectors; \( \alpha \) and \( \beta \) are scalars; and
bsrmv() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Several comments on bsrmv():

- Only blockDim > 1 is supported
- Only CUSPARSE_OPERATION_NON_TRANSPOSE is supported, that is
  \( y = \alpha \cdot A \cdot x + \beta \cdot y \)
- Only CUSPARSE_MATRIX_TYPE_GENERAL is supported.
- The size of vector \( x \) should be \((nb \times blockDim)\) at least, and the size of vector \( y \) should be \((mb \times blockDim)\) at least; otherwise, the kernel may return CUSPARSE_STATUS_EXECUTION_FAILED because of an out-of-bounds array.

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

```c
// Suppose that A is m x n sparse matrix represented by CSR format,
// hx is a host vector of size n, and hy is also a host vector of size m.
// m and n are not multiple of blockDim.
// step 1: transform CSR to BSR with column-major order
int base, nnz;
int nnzb;
cusparseDirection_t dirA = CUSPARSE_DIRECTION_COLUMN;
int mb = (m + blockDim-1)/blockDim;
int nb = (n + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
cusparseXcsr2bsrNnz(handle, dirA, m, n,
descrA, csrRowPtrA, csrColIndA, blockDim,
descrC, bsrRowPtrC, &nnzb);
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseXcsr2bsr(handle, dirA, m, n,
descrA, csrValA, csrRowPtrA, csrColIndA, blockDim,
descrC, bsrValC, bsrRowPtrC, bsrColIndC);
// step 2: allocate vector x and vector y large enough for bsrmv
cudaMalloc((void**)&x, sizeof(float)*(nb*blockDim));
cudaMalloc((void**)&y, sizeof(float)*(mb*blockDim));
cudaMemcpy(x, hx, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(y, hy, sizeof(float)*m, cudaMemcpyHostToDevice);
// step 3: perform bsrmv
cusparseSbsrmv(handle, dirA, transA, mb, nb, nnzb, &alpha,
descrC, bsrValC, bsrRowPtrC, bsrColIndC, blockDim, x, &beta, y);
```

**Input**

| handle | handle to the cuSPARSE library context. |
| dir    | storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN. |
trans | the operation $\text{op}(A)$. Only CUSPARSE_OPERATION_NON_TRANSPOSE is supported.
---|---
mb | number of block rows of matrix $A$.
nb | number of block columns of matrix $A$.
nnz | number of nonzero blocks of matrix $A$.
alpha | $<$type$>$ scalar used for multiplication.
descr | the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.
bsrVal | $<$type$>$ array of $\text{nnz}(=\text{csrRowPtrA}(mb) - \text{csrRowPtrA}(0))$ nonzero blocks of matrix $A$.
bsrRowPtr | integer array of $mb + 1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColInd | integer array of $\text{nnz}(=\text{csrRowPtrA}(mb) - \text{csrRowPtrA}(0))$ column indices of the nonzero blocks of matrix $A$.
blockDim | block dimension of sparse matrix $A$, larger than zero.
x | $<$type$>$ vector of $nb \times \text{blockDim}$ elements.
beta | $<$type$>$ scalar used for multiplication. If $\beta$ is zero, $y$ does not have to be a valid input.
y | $<$type$>$ vector of $mb \times \text{blockDim}$ elements.

Output

$y$ | $<$type$>$ updated vector.

See `cusparseStatus_t` for the description of the return status

### 8.2. `cusparse<t>bsrxmlv()`

```c
cusparseStatus_t
cusparseSbsrxmlv(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 mb,  
int nb,  
int nnzb,  
const double* alpha,  
const cusparseMatDescr_t descr,  
const double* bsrVal,  
const int* bsrMaskPtr,  
const int* bsrRowPtr,  
const int* bsrEndPtr,  
int blockDim,  
const double* x,  
const double* beta,  
double* y)

cusparseStatus_t  
cusparseCbsrxmv(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* bsrMaskPtr,  
const int* bsrRowPtr,  
const int* bsrEndPtr,  
int blockDim,  
const cuComplex* x,  
const cuComplex* beta,  
cuComplex* y)

cusparseStatus_t  
cusparseZbsrxmv(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* bsrMaskPtr,  
const int* bsrRowPtr,  
const int* bsrEndPtr,
This function performs a bsrMV and a mask operation

\[ y(\text{mask}) = (\alpha \ast \text{op}(A) \ast x + \beta \ast y)(\text{mask}) \]

where \( A \) is an \((mb \ast blockDim) \times (nb \ast blockDim)\) sparse matrix that is defined in BSRX storage format by the four arrays \( \text{bsrVal} \), \( \text{bsrRowPtr} \), \( \text{bsrEndPtr} \), and \( \text{bsrColInd} \); \( x \) and \( y \) are vectors; \( \alpha \) and \( \beta \) are scalars; and

\[
\text{op}(A) = \begin{cases} A_1 & \text{if trans == CUSPARSE_OPERATION_NON_TRANSPOSE} \\ A_1^T & \text{if trans == CUSPARSE_OPERATION_TRANSPOSE} \\ A_1^{H} & \text{if trans == CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE} \end{cases}
\]

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

For example, consider the \( 2 \times 3 \) 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

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

Suppose we want to do the following bsrMV operation on a matrix \( \overline{A} \) which is slightly different from \( A \).

\[
\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} := \text{alpha} \ast \left( \begin{bmatrix} O & O & O \\ O & A_{22} & O \end{bmatrix} \right) \ast \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} y_1 \\ \beta \ast y_2 \end{bmatrix}
\]

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

For example, the following \( \text{bsrRowPtr} \) and \( \text{bsrEndPtr} \) can represent matrix \( \overline{A} \):

\[
\text{bsrRowPtr} = \begin{bmatrix} 1 & 4 \end{bmatrix} \\
\text{bsrEndPtr} = \begin{bmatrix} 1 & 5 \end{bmatrix}
\]

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

The mask operator is equivalent to the following operation:
If a block row is not present in the `bsrMaskPtr`, then no calculation is performed on that row, and the corresponding value in `y` is unmodified. The question mark “?” is used to indicate row blocks not in `bsrMaskPtr`.

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

\[
\begin{bmatrix}
? \\
\end{bmatrix} := \alpha \begin{bmatrix}
? & ? & ? \\
O & A_{22} & O \\
\end{bmatrix} \begin{bmatrix}
X_1 \\
X_2 \\
X_3 \\
\end{bmatrix} + \beta \begin{bmatrix}
? \\
y_2 \\
\end{bmatrix}
\]

`bsrRowPtr` has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

A couple of comments on `bsrRowPtr`:

- Only `blockDim > 1` is supported
- Only `CUSPARSE_OPERATION_NON_TRANSPOSE` and `CUSPARSE_MATRIX_TYPE_GENERAL` are supported.
- Parameters `bsrMaskPtr`, `bsrRowPtr`, `bsrEndPtr` and `bsrColInd` are consistent with base index, either one-based or zero-based. The above example is one-based.

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>dir</code></td>
<td>storage format of blocks, either <code>CUSPARSE_DIRECTION_ROW</code> or <code>CUSPARSE_DIRECTION_COLUMN</code>.</td>
</tr>
<tr>
<td><code>trans</code></td>
<td>the operation <code>op(A)</code>. Only <code>CUSPARSE_OPERATION_NON_TRANSPOSE</code> is supported.</td>
</tr>
<tr>
<td><code>sizeOfMask</code></td>
<td>number of updated block rows of <code>y</code>.</td>
</tr>
<tr>
<td><code>mb</code></td>
<td>number of block rows of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>nb</code></td>
<td>number of block columns of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>nnzb</code></td>
<td>number of nonzero blocks of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td><code>&lt;type&gt;</code> scalar used for multiplication.</td>
</tr>
<tr>
<td><code>descr</code></td>
<td>the descriptor of matrix <code>A</code>. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td><code>bsrVal</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnz</code> nonzero blocks of matrix <code>A</code>.</td>
</tr>
</tbody>
</table>
bsrMaskPtr | integer array of `sizeOfMask` elements that contains the indices corresponding to updated block rows.

bsrRowPtr | integer array of `mb` elements that contains the start of every block row.

bsrEndPtr | integer array of `mb` elements that contains the end of every block row plus one.

bsrColInd | integer array of `nnzb` column indices of the nonzero blocks of matrix `A`.

blockDim | block dimension of sparse matrix `A`, larger than zero.

x | `<type>` vector of `mb * blockDim` elements.

beta | `<type>` scalar used for multiplication. If `beta` is zero, `y` does not have to be a valid input.

y | `<type>` vector of `mb * blockDim` elements.

See `cusparseStatus_t` for the description of the return status

### 8.3. `cusparse<t>bsrsv2_bufferSize()`

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

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

```c
void cusparseCbsrsv2_bufferSize(cusparseHandle_t handle,
                                cusparseDirection_t dirA,
                                cusparseOperation_t transA,
                                int mb,
                                int nnzb,
                                const cusparseMatDescr_t descrA,
                                double* bsrValA,
                                const int* bsrRowPtrA,
                                const int* bsrColIndA,
                                int blockDim,
                                bsrsv2Info_t info,
                                int* pBufferSizeInBytes)
```
This function returns size of the buffer used in \( \text{bsrsv2} \), a new sparse triangular linear system \( \text{op}(A) * y = \alpha x \).

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

\[
\text{op}(A) = \begin{cases} 
  A & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
  A^T & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
  A^H & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE}
\end{cases}
\]

Although there are six combinations in terms of parameter \( \text{trans} \) and the upper (lower) triangular part of \( A \), \( \text{bsrsv2\_bufferSize()} \) returns the maximum size buffer among these combinations. The buffer size depends on the dimensions \( mb \), \( blockDim \), and the number of nonzero blocks of the matrix \( nnzb \). If the user changes the matrix, it is necessary to call \( \text{bsrsv2\_bufferSize()} \) again to have the correct buffer size; otherwise a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{handle} )</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( \text{dirA} )</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>( \text{transA} )</td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td>( mb )</td>
<td>number of block rows of matrix ( A ).</td>
</tr>
<tr>
<td>( nnzb )</td>
<td>number of nonzero blocks of matrix ( A ).</td>
</tr>
</tbody>
</table>
**descrA**
the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, while the supported diagonal types are CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.

**bsrValA**
<type> array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) nonzero blocks of matrix A.

**bsrRowPtrA**
integer array of mb + 1 elements that contains the start of every block row and the end of the last block row plus one.

**bsrColIndA**
integer array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) column indices of the nonzero blocks of matrix A.

**blockDim**
block dimension of sparse matrix A; must be larger than zero.

**Output**

<table>
<thead>
<tr>
<th>info</th>
<th>record of internal states based on different algorithms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pBufferSizeInBytes</td>
<td>number of bytes of the buffer used in the bsrsv2_analysis() and bsrsv2_solve().</td>
</tr>
</tbody>
</table>

See **cusparseStatus_t** for the description of the return status.

### 8.4. cusparse<t>bsrsv2_analysis()

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

```c
cusparseStatus_t cusparseDbsrsv2_analysis(cusparseHandle_t handle,
cusparseDirection_t dirA,
cusparseOperation_t transA,
int mb,
int nnzb,
const cusparseMatDescr_t descrA,
const double* bsrValA,
const int* bsrRowPtrA,
const int* bsrColIndA,
```
This function performs the analysis phase of \( \text{bsrsv2} \), a new sparse triangular linear system 
\[ \text{op}(A) * y = \alpha x. \]

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

\[ \text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE_OPERATION_NON_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE_OPERATION_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE} \end{cases} \]

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

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

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

Function \( \text{bsrsv2_analysis()} \) reports a structural zero and computes level information, which stored in the opaque structure \( \text{info} \). The level information can extract more parallelism.
for a triangular solver. However \texttt{bsrsv2\_solve()} can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}.

Function \texttt{bsrsv2\_analysis()} always reports the first structural zero, even when parameter \texttt{policy} is \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}. No structural zero is reported if \texttt{CUSPARSE\_DIAG\_TYPE\_UNIT} is specified, even if block \(A(j, j)\) is missing for some \(j\). The user needs to call \texttt{cusparseXbsrsv2\_zeroPivot()} to know where the structural zero is.

It is the user's choice whether to call \texttt{bsrsv2\_solve()} if \texttt{bsrsv2\_analysis()} reports a structural zero. In this case, the user can still call \texttt{bsrsv2\_solve()}, which will return a numerical zero at the same position as a structural zero. However the result \(x\) is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks, either \texttt{CUSPARSE_DIRECTION_ROW} or \texttt{CUSPARSE_DIRECTION_COLUMN}.</td>
</tr>
<tr>
<td>transA</td>
<td>the operation (\text{op}(A)).</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of matrix (A).</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix (A).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix (A). The supported matrix type is \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}, while the supported diagonal types are \texttt{CUSPARSE_DIAG_TYPE_UNIT} and \texttt{CUSPARSE_DIAG_TYPE_NON_UNIT}.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>(&lt;\text{type}&gt;) array of (\text{nnzb} = \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0)) nonzero blocks of matrix (A).</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of (\text{mb} + 1) elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of (\text{nnzb} = \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0)) column indices of the nonzero blocks of matrix (A).</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix (A), larger than zero.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using \texttt{cusparseCreateBsrsv2Info()}.</td>
</tr>
<tr>
<td>policy</td>
<td>the supported policies are \texttt{CUSPARSE_SOLVE_POLICY_NO_LEVEL} and \texttt{CUSPARSE_SOLVE_POLICY_USE_LEVEL}.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is return by \texttt{bsrsv2_bufferSize()}.</td>
</tr>
</tbody>
</table>
Output

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

See `cusparseStatus_t` for the description of the return status

### 8.5. `cusparse<t>bsrsv2_solve()`

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

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

```c
cusparseStatus_t cusparseCbsrsv2_solve(cusparseHandle_t handle,
                                        cusparseDirection_t dirA,
                                        cusparseOperation_t transA,
                                        int mb,
                                        int nnzb,
                                        const cuComplex* alpha,
                                        const cusparseMatDescr_t descrA,
                                        const cuComplex* bsrValA,
                                        const int* bsrRowPtrA,
                                        const int* bsrColIndA,
                                        int blockDim,
                                        bsrsv2Info_t info,
```
This function performs the solve phase of \texttt{bsrsv2}, a new sparse triangular linear system 
\[ \text{op}(A) \cdot y = \alpha \cdot x. \]

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

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

The block in BSR format is of size \( \text{blockDim} \times \text{blockDim} \), stored as column-major or row-major as determined by parameter \( \text{dirA} \), which is either \text{CUSPARSE\_DIRECTION\_COLUMN} or \text{CUSPARSE\_DIRECTION\_ROW}. The matrix type must be \text{CUSPARSE\_MATRIX\_TYPE\_GENERAL}, and the fill mode and diagonal type are ignored. Function \texttt{bsrsv02\_solve()} can support an arbitrary \text{blockDim}.

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

This function requires a buffer size returned by \texttt{bsrsv2\_bufferSize()}. The address of \( \text{pBuffer} \) must be multiple of 128 bytes. If it is not, \text{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

Although \texttt{bsrsv2\_solve()} can be done without level information, the user still needs to be aware of consistency. If \texttt{bsrsv2\_analysis()} is called with policy \text{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL}, \texttt{bsrsv2\_solve()} can be run with or without levels. On the other hand, if \texttt{bsrsv2\_analysis()} is called with \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}, \texttt{bsrsv2\_solve()} can only accept \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}; otherwise, \text{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

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

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

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

The function supports the following properties if pBuffer != NULL:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

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

```c
// Suppose that L is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// L is lower triangular with unit diagonal.
// Assumption:
// - dimension of matrix L is m(=mb*blockDim),
// - matrix L has nnz(=nnzb*blockDim*blockDim) nonzero elements,
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of L on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_x and d_y are of size m.

cusparseMatDescr_t descr = 0;
bsrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

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

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

// step 3: query how much memory used in bsrsv2, and allocate the buffer
cusparseDbsrsv2_bufferSize(handle, dir, trans, mb, nnzb, descr,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &pBufferSize);
// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

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

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

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

**Input**

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

Output
---|---
y | <type> solution vector of size m.
---|---

See cusparseStatus_t for the description of the return status

8.6. cusparseXbsrsv2_zeroPivot()

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

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

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

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

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

▶ The routine requires no extra storage
▶ The routine does not support asynchronous execution
▶ The routine does not support CUDA graph capture

Input
---|---
handle | handle to the cusPARSE library context.
---|---
info | info contains a structural zero or numerical zero if the user already called bsrsv2_analysis() or bsrsv2_solve().
---|---

Output
position

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

See `cusparseStatus_t` for the description of the return status.

### 8.7. `cusparseCsrmvEx()`

This function performs the matrix-vector operation

\[
y = \alpha \times \text{op}(A) \times x + \beta \times y
\]
A is an \( m \times n \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \); \( x \) and \( y \) are vectors;

The function `cusparseCsrmvEx_bufferSize` returns the size of the workspace needed by `cusparseCsrmvEx`.

The function has the following limitations:

- All pointers should be aligned with 128 bytes
- Only `CUSPARSE_OPERATION_NON_TRANSPOSE` operation is supported
- Only `CUSPARSE_MATRIX_TYPE_GENERAL` matrix type is supported
- Only `CUSPARSE_INDEX_BASE_ZERO` indexing is supported
- Half-precision is not supported
- The minimum GPU architecture supported is SM_53

The function has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input specifically required by `cusparseCsrmvEx`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{alg} )</td>
<td>Algorithm implementation for csrmv, see <code>cusparseAlgMode_t</code> for possible values.</td>
</tr>
<tr>
<td>( \text{alphatype} )</td>
<td>Data type of alpha.</td>
</tr>
<tr>
<td>( \text{csrValAtype} )</td>
<td>Data type of ( \text{csrValA} ).</td>
</tr>
<tr>
<td>( \text{xtype} )</td>
<td>Data type of ( x ).</td>
</tr>
<tr>
<td>( \text{betatype} )</td>
<td>Data type of ( \beta ).</td>
</tr>
<tr>
<td>( \text{ytype} )</td>
<td>Data type of ( y ).</td>
</tr>
<tr>
<td>( \text{executiontype} )</td>
<td>Data type used for computation.</td>
</tr>
<tr>
<td>( \text{bufferSizeInBytes} )</td>
<td>Pointer to a <code>size_t</code> variable, which will be assigned with the size of workspace needed by <code>cusparseCsrmvEx</code>.</td>
</tr>
<tr>
<td>( \text{buffer} )</td>
<td>Pointer to workspace buffer</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

### 8.8. `cusparse<t>csrsv2_bufferSize()`

```c

void cusparseScsrsv2_bufferSize(cusparseHandle_t handle, cusparseOperation_t \( \text{transA} \), int \( m \), int \( \text{nnz} \), const cusparseMatDescr_t descrA, float* \( \text{csrValA} \));
```

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

$A$ is an $m \times m$ sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`; $x$ and $y$ are the right-hand-side and the solution vectors; $\alpha$ is a scalar; and

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

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

- The routine requires no extra storage.
The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

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

**Output**

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

See **cusparseStatus_t** for the description of the return status.

### 8.9. **cusparse<t>csrsv2_analysis()**

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

```c
 cusparseStatus_t
 cusparseDcsrsv2_analysis(cusparseHandle_t handle,                     
                         cusparseOperation_t transA,                  
                         int m,                                      
                         int nnz,                                    
                         const cusparseMatDescr_t descrA,            
                         const float* csrValA,                      
                         const int* csrRowPtrA,                     
                         const int* csrColIndA,                     
                         cusparseSolvePolicy_t policy,              
                         cusparseSolveInfo_t info,                  
                         void* pBuffer)                             
```
This function performs the analysis phase of `csrsv2`, a new sparse triangular linear system
\[ \text{op}(A) * y = \alpha x. \]

\( \text{A} \) is an \(m \times m\) sparse matrix that is defined in CSR storage format by the three arrays \(\text{csrValA}, \text{csrRowPtrA}, \text{and csrColIndA}\); \(x\) and \(y\) are the right-hand-side and the solution vectors; \(\alpha\) is a scalar; and

\[
\text{op}(A) = \begin{cases} 
  A & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
  A^T & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
  A^H & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE}
\end{cases}
\]

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

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

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

```c

int m, nnz,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
csrsv2Info_t info,
cusparseSolvePolicy_t policy,
void* pBuffer)

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

cusparseStatus_t
cusparseZcsrsv2_analysis(cusparseHandle_t         handle,
cusparseOperation_t      transA,
int                      m,
int                      nnz,
const cusparseMatDescr_t descrA,
const cuDoubleComplex*   csrValA,
const int*              csrRowPtrA,
const int*               csrColIndA,
csrsv2Info_t             info,
cusparseSolvePolicy_t    policy,
void*                    pBuffer)
```
Function `csrsv2_analysis()` always reports the first structural zero, even if the policy is `CUSPARSE_SOLVE_POLICY_NO_LEVEL`. No structural zero is reported if `CUSPARSE_DIAG_TYPE_UNIT` is specified, even if \( A(j,j) \) is missing for some \( j \). The user needs to call `cusparseXcsrsv2_zeroPivot()` to know where the structural zero is.

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

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>transA</td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>, while the supported diagonal types are <code>CUSPARSE_DIAG_TYPE_UNIT</code> and <code>CUSPARSE_DIAG_TYPE_NON_UNIT</code>.</td>
</tr>
<tr>
<td>csrValA</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of ( m + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using <code>cusparseCreateCsrsv2Info()</code>.</td>
</tr>
<tr>
<td>policy</td>
<td>The supported policies are <code>CUSPARSE_SOLVE_POLICY_NO_LEVEL</code> and <code>CUSPARSE_SOLVE_POLICY_USE_LEVEL</code>.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is returned by <code>csrsv2_bufferSize()</code>.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>structure filled with information collected during the analysis phase [that should be passed to the solve phase unchanged].</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
### 8.10. `cusparse<t>csrsv2_solve()`

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cusparseStatus_t</code></td>
<td><code>cusparseStatus_t</code></td>
</tr>
<tr>
<td><code>cusparseScsrsv2_solve</code></td>
<td><code>cusparseScsrsv2_solve</code></td>
</tr>
<tr>
<td><code>handle</code>, <code>transA</code>, <code>m</code>, <code>nnz</code>, <code>alpha</code>, <code>descra</code>, <code>csrValA</code>, <code>csrRowPtrA</code>, <code>csrColIndA</code>, <code>info</code>, <code>x</code>, <code>y</code>, <code>policy</code>, <code>pBuffer</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseDcsrsv2_solve</code></td>
<td><code>cusparseDcsrsv2_solve</code></td>
</tr>
<tr>
<td><code>handle</code>, <code>transA</code>, <code>m</code>, <code>nnz</code>, <code>alpha</code>, <code>descra</code>, <code>csrValA</code>, <code>csrRowPtrA</code>, <code>csrColIndA</code>, <code>info</code>, <code>x</code>, <code>y</code>, <code>policy</code>, <code>pBuffer</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseCcsrsv2_solve</code></td>
<td><code>cusparseCcsrsv2_solve</code></td>
</tr>
<tr>
<td><code>handle</code>, <code>transA</code>, <code>m</code>, <code>nnz</code>, <code>alpha</code>, <code>descra</code>, <code>csrValA</code>, <code>csrRowPtrA</code>, <code>csrColIndA</code>, <code>info</code>, <code>x</code>, <code>y</code>, <code>policy</code>, <code>pBuffer</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseZcsrsv2_solve</code></td>
<td><code>cusparseZcsrsv2_solve</code></td>
</tr>
<tr>
<td><code>handle</code>, <code>transA</code>, <code>m</code>, <code>nnz</code>, <code>alpha</code>, <code>descra</code>, <code>csrValA</code>, <code>csrRowPtrA</code>, <code>csrColIndA</code>, <code>info</code>, <code>x</code>, <code>y</code>, <code>policy</code>, <code>pBuffer</code></td>
<td></td>
</tr>
</tbody>
</table>
This function performs the solve phase of `csrsv2`, a new sparse triangular linear system

\[ \text{op}(A) \cdot y = \alpha \cdot x. \]

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

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

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

This function requires the buffer size returned by `csrsv2_bufferSize()`. The address of `pBuffer` must be multiple of 128 bytes. If it is not, `CUSPARSE\_STATUS\_INVALID\_VALUE` is returned.

Although `csrsv2\_solve()` can be done without level information, the user still needs to be aware of consistency. If `csrsv2\_analysis()` is called with policy `CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL`, `csrsv2\_solve()` can be run with or without levels. On the contrary, if `csrsv2\_analysis()` is called with `CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL`, `csrsv2\_solve()` can only accept `CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL`; otherwise, `CUSPARSE\_STATUS\_INVALID\_VALUE` is returned.

The level information may not improve the performance but spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, `CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL` performs better than `CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL`. If the user has an iterative solver, the best approach is to do `csrsv2\_analysis()` with `CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL` once. Then do `csrsv2\_solve()` with `CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL` in the first run and with `CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL` in the second run, picking faster one to perform the remaining iterations.

Function `csrsv2\_solve()` reports the first numerical zero, including a structural zero. If status is 0, no numerical zero was found. Furthermore, no numerical zero is reported if `CUSPARSE\_DIAG\_TYPE\_UNIT` is specified, even if \( A(j,j) \) is zero for some \( j \). The user needs to call `cusparseXcsrsv2\_zero\_Pivot()` to know where the numerical zero is.

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

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

cusparseMatDescr_t descr = 0;
crsrv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;

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

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

// step 3: query how much memory used in csrsv2, and allocate the buffer

cusparseDcsrsv2_bufferSize(handle, trans, m, nnz, descr,
   d_csrVal, d_csrRowPtr, d_csrColInd, info, &pBufferSize);
// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

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

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

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

---

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

The function supports the following properties if pBuffer != NULL

- The routine requires no extra storage
- The routine supports asynchronous execution
The routine supports CUDA graph capture

**Input**

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

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>(&lt;\text{type}&gt;) solution vector of size (m).</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 8.11. `cusparseXcsrsv2_zeroPivot()`

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

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

The `position` can be 0-based or 1-based, the same as the matrix.
Function `cusparseXcsrsv2_zeroPivot()` is a blocking call. It calls `cudaDeviceSynchronize()` to make sure all previous kernels are done. The position can be in the host memory or device memory. The user can set the proper mode with `cusparseSetPointerMode()`.

- The routine requires no extra storage
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

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

**Output**

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

See `cusparseStatus_t` for the description of the return status.

### 8.12. `cusparse<t>gemvi()`

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

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

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

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


```c
int                 m,
int                 n,
int                 nnz,
int*                pBufferSize)

cusparseStatus_t
cusparseSgemvi(cusparseHandle_t     handle,
cusparseOperation_t  transA,
int m,
int n,
const float*         alpha,
const float*         A,
int lda,
int nnz,
const float*         x,
const int*           xInd,
const float*         beta,
float*               y,
cusparseIndexBase_t  idxBase,
void*                pBuffer)

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

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

cusparseStatus_t
cusparseZgemvi(cusparseHandle_t       handle,
cusparseOperation_t    transA,
int m,
int n,
const cuDoubleComplex* alpha,
const cuDoubleComplex* A,
int lda,
```
This function performs the matrix-vector operation

\[ y = \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y \]

\( A \) is an \( m \times n \) dense matrix and a sparse vector \( x \) that is defined in a sparse storage format by the two arrays \( xVal, xInd \) of length \( \text{nnz} \), and \( y \) is a dense vector; \( \alpha \) and \( \beta \) are scalars; and

\[
\text{op}(A) = \begin{cases} 
A & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
A^T & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
A^H & \text{if } \text{trans} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE}
\end{cases}
\]

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

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

2. Call the \( \text{gemvi()} \) function with the \( \text{cusparseOperation_t} \) parameter set to \( \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \) 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.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

The function \( \text{cusparse<t>gemvi_bufferSize()} \) returns size of buffer used in \( \text{cusparse<t>gemvi()} \)

**Input**

- **handle**: handle to the cuSPARSE library context.
- **trans**: the operation \( \text{op}(A) \).
- **m**: number of rows of matrix \( A \).
- **n**: number of columns of matrix \( A \).
- **alpha**: \(<\text{type}>\) scalar used for multiplication.
- **A**: the pointer to dense matrix \( A \).
- **lda**: size of the leading dimension of \( A \).
- **nnz**: number of nonzero elements of vector \( x \).
- **x**: \(<\text{type}>\) sparse vector of \( \text{nnz} \) elements of size \( n \) if \( \text{op}(A) = A \), and size \( m \) if \( \text{op}(A) = A^T \) or \( \text{op}(A) = A^H \).
- **xInd**: Indices of non-zero values in \( x \).
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td><code>&lt;type&gt;</code> scalar used for multiplication. If <code>beta</code> is zero, <code>y</code> does not have to be a valid input.</td>
</tr>
<tr>
<td>y</td>
<td><code>&lt;type&gt;</code> dense vector of <code>m</code> elements if <code>op(A) = A</code>, and <code>n</code> elements if <code>op(A) = A^T</code> or <code>op(A) = A^H</code>.</td>
</tr>
<tr>
<td>idxBase</td>
<td>0 or 1, for 0 based or 1 based indexing, respectively</td>
</tr>
<tr>
<td>pBufferSize</td>
<td>Number of elements needed the buffer used in <code>cusparse&lt;t&gt;gemvi()</code>.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>Working space buffer</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td><code>&lt;type&gt;</code> updated dense vector.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.
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 `csrsm2_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 `csrsm2Info_t` that has been initialized previously with a call to `cusparseCreateCsrsm2Info()`.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the `csrsm2Info_t` parameter by calling the appropriate `csrsm2_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 `csrsm2Info_t` parameter can be released by calling `cusparseDestroyCsrsm2Info()`.

### 9.1. `cusparse<t>bsrmm()`

```c
 cusparseStatus_t
cusparseSbsrmm(cusparseHandle_t handle,
               cusparseDirection_t dirA,
               cusparseOperation_t transA,
               cusparseOperation_t transB,
               int mb,
               int n,
               int kb,
               int nnzb,
               const float* alpha,
               const cusparseMatDescr_t descrA,
               const float* bsrValA,
               const int* bsrRowPtrA,
               const int* bsrColIndA,
               int blockDim,
```

const float* B, int ldb, const float* beta, float* C, int ldc)
cusparseStatus_t cusparseDbsrmm(cusparseHandle_t handle, cusparseDirection_t dirA, cusparseOperation_t transA, cusparseOperation_t transB, int mb, int n, int kb, int nnzb, const double* alpha, const cusparseMatDescr_t descrA, const double* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, const double* B, int ldb, const double* beta, double* C, int ldc)
cusparseStatus_t cusparseCbsrmm(cusparseHandle_t handle, cusparseDirection_t dirA, cusparseOperation_t transA, cusparseOperation_t transB, int mb, int n, int kb, int nnzb, const cuComplex* alpha, const cusparseMatDescr_t descrA, const cuComplex* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, const cuComplex* B, int ldb, const cuComplex* beta, cuComplex* C, int ldc)
cusparseStatus_t cusparseZbsrmm(cusparseHandle_t handle, cusparseDirection_t dirA, cusparseOperation_t transA, cusparseOperation_t transB, int mb, int n, int kb, int nnzb, const cuDoubleComplex* alpha, const cusparseMatDescr_t descrA, const cuDoubleComplex* bsrValA, const int* bsrRowPtrA,
This function performs one of the following matrix-matrix operations:

\[ C = \alpha \star \text{op}(A) \star \text{op}(B) + \beta \star C \]

\( A \) is an \( mb \times kb \) sparse matrix that is defined in BSR storage format by the three arrays \( \text{bsrValA} \), \( \text{bsrRowPtrA} \), and \( \text{bsrColIndA} \); \( B \) and \( C \) are dense matrices; \( \alpha \) and \( \beta \) are scalars; and

\[
\text{op}(A) = \begin{cases} 
A & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
A^T & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_TRANSPOSE (not supported)} \\
A^H & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE (not supported)} 
\end{cases}
\]

and

\[
\text{op}(B) = \begin{cases} 
B & \text{if } \text{transB} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
B^T & \text{if } \text{transB} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
B^H & \text{if } \text{transB} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE (not supported)} 
\end{cases}
\]

The function has the following limitations:

- Only \text{CUSPARSE_MATRIX_TYPE_GENERAL} matrix type is supported
- Only \text{blockDim} > 1 is supported

The motivation of \text{transpose}(B) is to improve memory access of matrix \( B \). The computational pattern of \( A \star \text{trans}^\top(B) \) with matrix \( B \) in column-major order is equivalent to \( A \star B \) with matrix \( B \) in row-major order.

In practice, no operation in an iterative solver or eigenvalue solver uses \( A \star \text{trans}^\top(B) \). However, we can perform \( A \star \text{trans}^\top(\text{trans}^\top(B)) \) which is the same as \( A \star B \). For example, suppose \( A \) is \( mb \times kb \), \( B \) is \( k \times n \) and \( C \) is \( m \times n \), the following code shows usage of \text{cusparseDbsrmm}().

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

Instead of using \( A \star B \), our proposal is to transpose \( B \) to \( B^\top \) by first calling \text{cublas<t>geam}(), and then to perform \( A \star \text{trans}^\top(B^\top) \).
// step 1: Bt := transpose(B)
const int m = mb*blockSize;
const int k = kb*blockSize;
double *Bt;
const int ldb_Bt = n; // leading dimension of Bt
cudaMalloc((void**)&Bt, sizeof(double)*ldb_Bt*k);
double one  = 1.0;
double zero = 0.0;
cublasSetPointerMode(cublas_handle, CUBLAS_POINTER_MODE_HOST);
cublasDgeam(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_T,
n, k, &one, B, int ldb_B, &zero, B, int ldb_B, Bt, ldb_Bt);

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

bsrmm() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

| handle | handle to the cuSPARSE library context. |
| dir | storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN. |
| transA | the operation op(A). |
| transB | the operation op(B). |
| mb | number of block rows of sparse matrix A. |
| n | number of columns of dense matrix op(B) and A. |
| kb | number of block columns of sparse matrix A. |
| nnzb | number of non-zero blocks of sparse matrix A. |
| alpha | <type> scalar used for multiplication. |
| descrA | the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE. |
| bsrValA | <type> array of nnzb (= bsrRowPtrA(mb) – bsrRowPtrA(0)) nonzero blocks of matrix A. |
| bsrRowPtrA | integer array of mb + 1 elements that contains the start of every block row and the end of the last block row plus one. |
### cuSPARSE Level 3 Function Reference

#### cuSPARSE Library

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) column indices of the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A, larger than zero.</td>
</tr>
<tr>
<td>B</td>
<td>array of dimensions (ldb, n) if op(B)=B and (ldb, k) otherwise.</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of B. If op(B)=B, it must be at least max(1, k). If op(B) != B, it must be at least max(1, n).</td>
</tr>
<tr>
<td>beta</td>
<td>&lt;type&gt; scalar used for multiplication. If beta is zero, C does not have to be a valid input.</td>
</tr>
<tr>
<td>C</td>
<td>array of dimensions (ldc, n).</td>
</tr>
<tr>
<td>ldc</td>
<td>leading dimension of C. It must be at least max(1, m) if op(A)=A and at least max(1, k) otherwise.</td>
</tr>
</tbody>
</table>

**Output**

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

See [cusparseStatus_t](#) for the description of the return status

### 9.2. cusparse<t>bsrsm2_bufferSize()

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

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

This function returns size of buffer used in `bsrsm2()`, a new sparse triangular linear system

\[ op(A) \times op(X) = op(B) \].

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

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

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

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture
## Input

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

## Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>info</strong></td>
<td>record internal states based on different algorithms.</td>
</tr>
<tr>
<td><strong>pBufferSizeInBytes</strong></td>
<td>number of bytes of the buffer used in bsrsm2_analysis() and bsrsm2_solve().</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](https://docs.nvidia.com/cusparse/index.html) for the description of the return status

### 9.3. cusparse<t>bsrsm2_analysis()
This function performs the analysis phase of `bsrsm2()`, a new sparse triangular linear system
\( \text{op}(A) \ast \text{op}(X) = \alpha \text{op}(B) \).
A is an \((mb \times blockDim) \times (mb \times blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays \(bsrValA, bsrRowPtrA,\) and \(bsrColIndA\); \(B\) and \(X\) are the right-hand-side and the solution matrices; \(\alpha\) is a scalar; and

\[
\begin{align*}
\text{op}(A) &= \begin{cases} 
A & \text{if trans} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
A^T & \text{if trans} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
A^H & \text{if trans} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE}
\end{cases}
\end{align*}
\]

and

\[
\begin{align*}
\text{op}(X) &= \begin{cases} 
X & \text{if transX} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
X^T & \text{if transX} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
X^H & \text{if transX} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE (not supported)}
\end{cases}
\end{align*}
\]

and \(\text{op}(B)\) and \(\text{op}(X)\) are equal.

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

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

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

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

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

If \(\text{bsrsm2_analysis()}\) reports a structural zero, the solve will return a numerical zero in the same position as the structural zero but this result \(X\) is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks, either \text{CUSPARSE_DIRECTION_ROW} or \text{CUSPARSE_DIRECTION_COLUMN}.</td>
</tr>
<tr>
<td>transA</td>
<td>the operation (\text{op}(A)).</td>
</tr>
<tr>
<td>transX</td>
<td>the operation (\text{op}(B)) and (\text{op}(X)).</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of matrix (A).</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix op(B) and op(X).</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of non-zero blocks of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, while the supported diagonal types are CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of mb + 1 elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) column indices of the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A; larger than zero.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using cusparseCreateBsrsm2Info.</td>
</tr>
<tr>
<td>policy</td>
<td>The supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user; the size is return by bsrsm2_bufferSize().</td>
</tr>
</tbody>
</table>

**Output**

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

See cusparseStatus_t for the description of the return status

### 9.4. cusparse<t>bsrsm2_solve()

cusparseStatus_t cusparseSbsrsm2_solve(cusparseHandle_t handle, 
cusparseDirection_t dirA, 
cusparseOperation_t transA, 
cusparseOperation_t transX, 
int mb, 
int n, 
nint nnzb, 
const float* alpha, 
const cusparseMatDescr_t descrA, 
const float* bsrSortedVal, 
const int* bsrSortedRowPtr, 
const int* bsrSortedColInd, 
int blockDim, ...
cuSPARSE Level 3 Function Reference

```c
bsrsm2Info_t info,
const float* B,
int ldb,
float* X,
int ldx,
cusparseSolvePolicy_t policy,
void* pBuffer)
```

cusparseStatus_t cusparseDbsrsm2_solve(cusparseHandle_t handle,
cusparseDirection_t dirA,
cusparseOperation_t transA,
cusparseOperation_t transX,
int mb,
int n,
int nnzb,
const double* alpha,
const cusparseMatDescr_t descrA,
const double* bsrSortedVal,
const int* bsrSortedRowPtr,
const int* bsrSortedColInd,
int blockDim,
bsrsm2Info_t info,
const double* B,
int ldb,
double* X,
int ldx,
cusparseSolvePolicy_t policy,
void* pBuffer)
```

cusparseStatus_t cusparseCbsrsm2_solve(cusparseHandle_t handle,
cusparseDirection_t dirA,
cusparseOperation_t transA,
cusparseOperation_t transX,
int mb,
int n,
int nnzb,
const cuComplex* alpha,
const cusparseMatDescr_t descrA,
const cuComplex* bsrSortedVal,
const int* bsrSortedRowPtr,
const int* bsrSortedColInd,
int blockDim,
bsrsm2Info_t info,
const cuComplex* B,
int ldb,
cuComplex* X,
int ldx,
cusparseSolvePolicy_t policy,
void* pBuffer)
```

cusparseStatus_t cusparseZbsrsm2_solve(cusparseHandle_t handle,
cusparseDirection_t dirA,
cusparseOperation_t transA,
cusparseOperation_t transX,
int mb,
int n,
int nnzb,
const cuDoubleComplex* alpha,
This function performs the solve phase of the solution of a sparse triangular linear system:

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

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

\[
\text{op}(A) = \begin{cases} 
A & \text{if transA == CUSPARSE_OPERATION_NON_TRANSPOSE} \\
A^T & \text{if transA == CUSPARSE_OPERATION_TRANSPOSE} \\
A^H & \text{if transA == CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE} 
\end{cases}
\]

and

\[
\text{op}(X) = \begin{cases} 
X & \text{if transX == CUSPARSE_OPERATION_NON_TRANSPOSE} \\
X^T & \text{if transX == CUSPARSE_OPERATION_TRANSPOSE} \\
X^H & \text{not supported} 
\end{cases}
\]

Only \(\text{op}(A) = A\) is supported.

\(\text{op}(B)\) and \(\text{op}(X)\) must be performed in the same way. In other words, if \(\text{op}(B) = B\), \(\text{op}(X) = X\).

The block of BSR format is of size \(blockDim \times blockDim\), stored as column-major or row-major as determined by parameter \(dirA\), which is either \(CUSPARSE_DIRECTION_ROW\) or \(CUSPARSE_DIRECTION_COLUMN\). The matrix type must be \(CUSPARSE_MATRIX_TYPE_GENERAL\), and the fill mode and diagonal type are ignored. Function \(bsrsm2_solve()\) can support an arbitrary \(blockDim\).

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

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

Although \(bsrsm2_solve()\) can be done without level information, the user still needs to be aware of consistency. If \(bsrsm2_analysis()\) is called with policy \(CUSPARSE_SOLVE_POLICY_USE_LEVEL\), \(bsrsm2_solve()\) can be run with or without levels. On the other hand, if \(bsrsm2_analysis()\) is called with \(CUSPARSE_SOLVE_POLICY_NO_LEVEL\), \(bsrsm2_solve()\) can only accept \(CUSPARSE_SOLVE_POLICY_NO_LEVEL\); otherwise, \(CUSPARSE_STATUS_INVALID_VALUE\) is returned.

Function \(bsrsm2_solve()\) has the same behavior as \(bsrsv2_solve()\), reporting the first numerical zero, including a structural zero. The user must call \(cusparseXbsrsm2_query_zero_pivot()\) to know where the numerical zero is.
The motivation of \( \text{transpose}(X) \) is to improve the memory access of matrix \( X \). The computational pattern of \( \text{transpose}(X) \) with matrix \( X \) in column-major order is equivalent to \( X \) with matrix \( X \) in row-major order.

In-place is supported and requires that \( B \) and \( X \) point to the same memory block, and \( ldb = ldx \).

The function supports the following properties if \( pBuffer \neq NULL \):

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>transA</td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td>transX</td>
<td>the operation ( \text{op}(B) ) and ( \text{op}(X) ).</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of matrix ( A ).</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix ( \text{op}(B) ) and ( \text{op}(X) ).</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of non-zero blocks of matrix ( A ).</td>
</tr>
<tr>
<td>alpha</td>
<td>(&lt;\text{type}&gt;) scalar used for multiplication.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, while the supported diagonal types are CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnzb} = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0) ) non-zero blocks of matrix ( A ).</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of ( mb + 1 ) elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of ( \text{nnzb} = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0) ) column indices of the nonzero blocks of matrix ( A ).</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix ( A ); larger than zero.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using ( \text{cusparseCreateBsrsm2Info}() ).</td>
</tr>
<tr>
<td>B</td>
<td>(&lt;\text{type}&gt;) right-hand-side array.</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of ( B ). If ( \text{op}(B) = B ), ( ldb \geq (mb*blockDim) ); otherwise, ( ldb \geq n ).</td>
</tr>
<tr>
<td>ldx</td>
<td>leading dimension of ( X ). If ( \text{op}(X) = X ), then ( ldx \geq (mb*blockDim) ). Otherwise ( ldx \geq n ).</td>
</tr>
<tr>
<td>policy</td>
<td>the supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user; the size is returned by bsrsm2_bufferSize().</td>
</tr>
</tbody>
</table>

Output

| X | <type> solution array with leading dimensions ldx. |

See cusparseStatus_t for the description of the return status

9.5. cusparseXbsrsm2_zeroPivot()

cusparseStatus_t cusparseXbsrsm2_zeroPivot(cusparseHandle_t handle, bsrsm2Info_t info, int* position)

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

The position can be 0-base or 1-base, the same as the matrix. Function cusparseXbsrsm2_zeroPivot() is a blocking call. It calls cudaDeviceSynchronize() to make sure all previous kernels are done.

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

- The routine requires no extra storage
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

Input

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>info contains a structural zero or a numerical zero if the user already called bsrsm2_analysis() or bsrsm2_solve().</td>
</tr>
</tbody>
</table>

Output

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

See cusparseStatus_t for the description of the return status
9.6. \texttt{cusparse<t>csrsm2_bufferSizeExt()} \parskip=0pt

\begin{tabular}{|l|l|}
\hline
\texttt{cusparseStatus_t} & \texttt{cusparseStatus_t} \texttt{cusparseStatus_t} \\
\texttt{cusparseScsrsm2_bufferSizeExt(cusparseHandle_t handle,} & \texttt{cusparseScsrsm2_bufferSizeExt(cusparseHandle_t handle,} & \texttt{cusparseScsrsm2_bufferSizeExt(cusparseHandle_t handle,} \\
\texttt{int algo,} & \texttt{int algo,} & \texttt{int algo,} \\
\texttt{cusparseOperation_t transA,} & \texttt{cusparseOperation_t transA,} & \texttt{cusparseOperation_t transA,} \\
\texttt{cusparseOperation_t transB,} & \texttt{cusparseOperation_t transB,} & \texttt{cusparseOperation_t transB,} \\
\texttt{int m,} & \texttt{int m,} & \texttt{int m,} \\
\texttt{int nrhs,} & \texttt{int nrhs,} & \texttt{int nrhs,} \\
\texttt{int nnz,} & \texttt{int nnz,} & \texttt{int nnz,} \\
\texttt{const float* alpha,} & \texttt{const float* alpha,} & \texttt{const float* alpha,} \\
\texttt{const cusparseMatDescr_t descrA,} & \texttt{const cusparseMatDescr_t descrA,} & \texttt{const cusparseMatDescr_t descrA,} \\
\texttt{const float* csrSortedValA,} & \texttt{const float* csrSortedValA,} & \texttt{const float* csrSortedValA,} \\
\texttt{const int* csrSortedRowPtrA,} & \texttt{const int* csrSortedRowPtrA,} & \texttt{const int* csrSortedRowPtrA,} \\
\texttt{const int* csrSortedColIndA,} & \texttt{const int* csrSortedColIndA,} & \texttt{const int* csrSortedColIndA,} \\
\texttt{const float* B,} & \texttt{const float* B,} & \texttt{const float* B,} \\
\texttt{int ldb,} & \texttt{int ldb,} & \texttt{int ldb,} \\
\texttt{csrsm2Info_t info,} & \texttt{csrsm2Info_t info,} & \texttt{csrsm2Info_t info,} \\
\texttt{cusparseSolvePolicy_t policy,} & \texttt{cusparseSolvePolicy_t policy,} & \texttt{cusparseSolvePolicy_t policy,} \\
\texttt{size_t* pBufferSize)} & \texttt{size_t* pBufferSize)} & \texttt{size_t* pBufferSize)} \\
\hline
\end{tabular}
This function returns the size of the buffer used in \texttt{csrsm2}, a sparse triangular linear system \( \text{op}(A) \times \text{op}(X) = \text{op}(B) \).

\( A \) is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \); \( B \) and \( X \) are the right-hand-side matrix and the solution matrix; \( \alpha \) is a scalar; and

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

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{handle}</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>\texttt{algo}</td>
<td>algo = 0 is non-block version; algo = 1 is block version.</td>
</tr>
<tr>
<td>\texttt{transA}</td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td>\texttt{transB}</td>
<td>the operation ( \text{op}(B) ).</td>
</tr>
<tr>
<td>\texttt{m}</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{nrhs}</td>
<td>number of columns of right hand side matrix ( \text{op}(B) ).</td>
</tr>
<tr>
<td>\texttt{nnz}</td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{alpha}</td>
<td>(&lt;\text{type}&gt;) scalar used for multiplication.</td>
</tr>
<tr>
<td>\texttt{descrA}</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, while the supported diagonal types are CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.</td>
</tr>
</tbody>
</table>
### csrValA

A `<type>` array of `nnz` elements. `nnz` is given by `csrRowPtrA(m) - csrRowPtrA(0)`. It contains the nonzero elements of matrix `A`.

### csrRowPtrA

An integer array of `m + 1` elements that contains the start of every row and the end of the last row plus one.

### csrColIndA

An integer array of `nnz` elements. `nnz` is given by `csrRowPtrA(m) - csrRowPtrA(0)`. It contains the column indices of the nonzero elements of matrix `A`.

### B

A `<type>` right-hand-side matrix. `op(B)` is of size `m`-by-`nrhs`.

### ldb

Leading dimension of `B` and `X`.

### info

A structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).

### policy

The supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.

### Output

#### info

A record of internal states based on different algorithms.

#### pBufferSize

Number of bytes of the buffer used in the `csrsm2_analysis` and `csrsm2_solve`.

---

See **cusparseStatus_t** for the description of the return status.

### 9.7. **cusparse<t>csrsm2_analysis()**

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

```c
cusparseStatus_t
cusparseDcsrsm2_analysis(cusparseHandle_t handle,
                        int algo,
                        cusparseOperation_t transA,
                        cusparseOperation_t transB,
                        int m,
                        int nrhs,
                        int nnz,
                        const float* alpha,
                        const cusparseMatDescr_t descrA,
                        const float* csrSortedValA,
                        const int* csrSortedRowPtrA,
                        const int* csrSortedColIndA,
                        const float* B,
                        int ldb,
                        csrsm2Info_t info,
                        cusparseSolvePolicy_t policy,
                        void* pBuffer)
```
This function performs the analysis phase of `csrsm2`, a sparse triangular linear system \( \alpha \cdot \text{op}(A) \cdot \text{op}(X) = \text{op}(B) \).

A is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`; \( B \) and \( X \) are the right-hand-side matrix and the solution matrix; \( \alpha \) is a scalar; and
It is expected that this function will be executed only once for a given matrix and a particular operation type.

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

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

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

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

**Input**

- **handle** handle to the cuSPARSE library context.
- **algo** algo = 0 is non-block version; algo = 1 is block version.
- **transA** the operation \( \text{op}(A) \).
- **transB** the operation \( \text{op}(B) \).
- **m** number of rows of matrix \( A \).
- **nrhs** number of columns of matrix \( \text{op}(B) \).
- **nnz** number of nonzero elements of matrix \( A \).
- **alpha** \(<\text{type}>\) scalar used for multiplication.
- **descrA** the descriptor of matrix \( A \). The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`, while the supported diagonal types are `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.
- **csrValA** \(<\text{type}>\) array of \( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) \) nonzero elements of matrix \( A \).
- **csrRowPtrA** integer array of \( m + 1 \) elements that contains the start of every row and the end of the last row plus one.
- **csrColIndA** integer array of \( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) \) column indices of the nonzero elements of matrix \( A \).
**B**

<Type> right-hand-side matrix. op(B) is of size m-by-nrhs.

**ldb**

Leading dimension of B and X.

**info**

Structure initialized using cusparseCreateCsrsv2Info().

**policy**

The supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and
CUSPARSE_SOLVE_POLICY_USE_LEVEL.

** pBuffer**

Buffer allocated by the user, the size is returned by csrsm2_bufferSize().

---

**Output**

**info**

Structure filled with information collected during the analysis phase [that should be passed to the solve phase unchanged].

See **cusparseStatus_t** for the description of the return status.

### 9.8. **cusparse<	>csrsm2_solve()**

```c
#define HANDLE(handle, algo, transA, transB, m, nrhs, nnz, alpha, descrA, csrSortedValA, csrSortedRowPtrA, csrSortedColIndA, B, ldb, info, policy, pBuffer)

cusparseStatus_t
cusparseScsrsm2_solve(cusparseHandle_t handle, algo, transA, transB, m, nrhs, nnz, alpha, descrA, csrSortedValA, csrSortedRowPtrA, csrSortedColIndA, B, ldb, info, policy, pBuffer)

cusparseStatus_t
cusparseDcsrsm2_solve(cusparseHandle_t handle, algo, transA, transB, m, nrhs, nnz, alpha, descrA, csrSortedValA, csrSortedRowPtrA, csrSortedColIndA, B, ldb, info, policy, pBuffer)
```

---

**cuSPARSE Library**

DU-06709-001_v11.2 | 91
This function performs the solve phase of `csrsm2`, a sparse triangular linear system \( \text{op}(A) \cdot \text{op}(X) = \text{op}(B) \).

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

\[
\text{op}(A) = \begin{cases} 
  A & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\
  A^T & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_TRANSPOSE} \\
  A^H & \text{if } \text{transA} == \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE}
\end{cases}
\]

\( \text{transB} \) acts on both matrix \( B \) and matrix \( X \), only \text{CUSPARSE_OPERATION_NON_TRANSPOSE} and \text{CUSPARSE_OPERATION_TRANSPOSE}. The operation is in-place, matrix \( B \) is overwritten by matrix \( X \).

\( \text{ldb} \) must be not less than \( m \) if \( \text{transB} = \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \). Otherwise, \( \text{ldb} \) must be not less than \( \text{nrhs} \).
This function requires the buffer size returned by `csrsm2_bufferSize()`. The address of `pBuffer` must be multiple of 128 bytes. If it is not, `CUSPARSE_STATUS_INVALID_VALUE` is returned.

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

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

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

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

Appendix section shows an example of `csrsm2`.

The function supports the following properties if `pBuffer` != `NULL`

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>algo</code></td>
<td><code>algo = 0</code> is non-block version; <code>algo = 1</code> is block version.</td>
</tr>
<tr>
<td><code>transA</code></td>
<td>the operation <code>op(A)</code>.</td>
</tr>
<tr>
<td><code>transB</code></td>
<td>the operation <code>op(B)</code>.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows and columns of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>nrhs</code></td>
<td>number of columns of matrix <code>op(B)</code>.</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>number of nonzeros of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td><code>&lt;type&gt;</code> scalar used for multiplication.</td>
</tr>
</tbody>
</table>
The descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`, while the supported diagonal types are `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.

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

- **csrRowPtrA**: integer array of `m + 1` elements that contains the start of every row and the end of the last row plus one.

- **csrColIndA**: integer array of `nnz = csrRowPtrA(m) - csrRowPtrA(0)` column indices of the nonzero elements of matrix A.

- **B**: `<type>` right-hand-side matrix. `op(B)` is of size `m-by-nrhs`.

- **ldb**: leading dimension of B and X.

- **info**: structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).

- **policy**: The supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.

- **pBuffer**: buffer allocated by the user, the size is returned by `csrsm2_bufferSize`.

### Output

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

See `cusparseStatus_t` for the description of the return status.

### 9.9. cusparseXcsrsm2_zeroPivot()

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

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

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

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

The position can be in the host memory or device memory. The user can set the proper mode with `cusparseSetPointerMode()`.
The routine requires no extra storage

- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>info contains structural zero or numerical zero if the user already called csrsm2_analysis() or csrsm2_solve().</td>
</tr>
</tbody>
</table>

**Output**

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

See [cusparseStatus_t](#) for the description of the return status

### 9.10. cusparse<t>gemmi() [DEPRECATED]

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

```c
cusparseStatus_t
cusparseSgemmi(cusparseHandle_t handle,
              int              m,
              int              n,
              int              k,
              int              nnz,
              const float*    alpha,
              const float*    A,
              int              lda,
              const float*    cscValB,
              const int*      cscColPtrB,
              const int*      cscRowIndB,
              const float*    beta,
              float*          C,
              int              ldc)

cusparseStatus_t
cusparseDgemmi(cusparseHandle_t handle,
              int              m,
              int              n,
              int              k,
              int              nnz,
              const double*   alpha,
              const double*   A,
              int              lda,
              const double*   cscValB,
              const int*      cscColPtrB,
              const int*      cscRowIndB,
              const double*   beta,
              double*         C,
```

---

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

\[ C = \alpha \cdot A \cdot B + \beta \cdot C \]

A and C are dense matrices; B is a \( k \times n \) sparse matrix that is defined in CSC storage format by
the three arrays `cscValB`, `cscColPtrB`, and `cscRowIndB`; \( \alpha \) and \( \beta \) are scalars; and

Remark: \( B \) is base-0.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrices B and C.</td>
</tr>
<tr>
<td>k</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzero elements of sparse matrix B.</td>
</tr>
<tr>
<td>alpha</td>
<td>&lt;type&gt; scalar used for multiplication.</td>
</tr>
<tr>
<td>A</td>
<td>array of dimensions ((l_{da}, k)).</td>
</tr>
<tr>
<td><strong>lda</strong></td>
<td>leading dimension of A. It must be at least m</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><strong>cscValB</strong></td>
<td><code>&lt;type&gt;</code> array of ( \text{nnz}(= \text{cscColPtrB}(k) - \text{cscColPtrB}(0)) ) nonzero elements of matrix B.</td>
</tr>
<tr>
<td><strong>cscColPtrB</strong></td>
<td>integer array of ( k + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td><strong>cscRowIndB</strong></td>
<td>integer array of ( \text{nnz}(= \text{cscColPtrB}(k) - \text{cscColPtrB}(0)) ) column indices of the nonzero elements of matrix B.</td>
</tr>
<tr>
<td><strong>beta</strong></td>
<td><code>&lt;type&gt;</code> scalar used for multiplication. If <code>beta</code> is zero, <code>C</code> does not have to be a valid input.</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>array of dimensions ((ldc, n)).</td>
</tr>
<tr>
<td><strong>ldc</strong></td>
<td>leading dimension of C. It must be at least m</td>
</tr>
</tbody>
</table>

**Output**

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

See [cusparseStatus_t](#) for the description of the return status
Chapter 10. cuSPARSE Extra Function Reference

This chapter describes the extra routines used to manipulate sparse matrices.

10.1. cusparse<t>csrgem2()

cusparseStatus_t
cusparseScsrgem2_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const float* alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const float* const int* const int* const float* const cusparseMatDescr_t descrB,
int nnzB,
const float* const int* const int* const float* const cusparseMatDescr_t descrC,
const int* const int* size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrgem2_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const double* alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const double* const int* const int* const double* const cusparseMatDescr_t descrB,
int nnzB,
const double* const int* const int* const double* const cusparseMatDescr_t descrC,
const int* const int*
const int*               csrSortedColIndB,
const cusparseMatDescr_t descrC,
const double*             csrSortedValC,
const int*               csrSortedRowPtrC,
const int*               csrSortedColIndC,
size_t*                  pBufferSizeInBytes)

 cusparseStatus_t
 cusparseCcsrgeam2_bufferSizeExt(cusparseHandle_t         handle,
 int
 int
 const cuComplex*         alpha,
 const cusparseMatDescr_t descrA,
 int
 const cuComplex*         csrSortedValA,
 const int*               csrSortedRowPtrA,
 const int*               csrSortedColIndA,
 const cuComplex*         beta,
 const cusparseMatDescr_t descrB,
 int
 const cuComplex*         csrSortedValB,
 const int*               csrSortedRowPtrB,
 const int*               csrSortedColIndB,
 const cuComplex*         csrSortedColIndC,
 const int*               csrSortedRowPtrC,
 const int*               csrSortedColIndC,
 size_t*                  pBufferSizeInBytes)

 cusparseStatus_t
 cusparseZcsrgeam2_bufferSizeExt(cusparseHandle_t         handle,
 int
 int
 const cuDoubleComplex*   alpha,
 const cusparseMatDescr_t descrA,
 int
 const cuDoubleComplex*   csrSortedValA,
 const int*               csrSortedRowPtrA,
 const int*               csrSortedColIndA,
 const cuDoubleComplex*   beta,
 const cusparseMatDescr_t descrB,
 int
 const cuDoubleComplex*   csrSortedValB,
 const int*               csrSortedRowPtrB,
 const int*               csrSortedColIndB,
 const cuDoubleComplex*   csrSortedColIndC,
 const int*               csrSortedRowPtrC,
 const int*               csrSortedColIndC,
 size_t*                  pBufferSizeInBytes)

 cusparseStatus_t
 cusparseXcsrgeam2Nnz(cusparseHandle_t         handle,
 int
 int
 const cusparseMatDescr_t descrA,
 int
 const int*               csrSortedRowPtrA,
 const int*               csrSortedColIndA,
 const cusparseMatDescr_t descrB,
 int
 const cuDoubleComplex*   csrSortedValB,
 const int*               csrSortedRowPtrB,
 const int*               csrSortedColIndB,
 const cuDoubleComplex*   csrSortedColIndC,
 const int*               csrSortedRowPtrC,
 const int*               csrSortedColIndC,
 size_t*                  pBufferSizeInBytes)
**cuSPARSE Extra Function Reference**

---

**cuSPARSE Library**

---

```c
const int*               csrSortedRowPtrB,
const int*               csrSortedColIndB,
const cusparseMatDescr_t descrC,
int*                     csrSortedRowPtrC,
int*                     nnzTotalDevHostPtr,
void*                    workspace)
```

```c
cusparseStatus_t
cusparseScsrgem2(cusparseHandle_t         handle,
int                      m,
int                      n,
const float*             alpha,
const cusparseMatDescr_t descrA,
int                      nnzA,
const float*             csrSortedValA,
const int*               csrSortedRowPtrA,
const int*               csrSortedColIndA,
const float*             beta,
const cusparseMatDescr_t descrB,
int                      nnzB,
const float*             csrSortedValB,
const int*               csrSortedRowPtrB,
const int*               csrSortedColIndB,
const cusparseMatDescr_t descrC,
float*                   csrSortedValC,
int*                     csrSortedRowPtrC,
int*                     csrSortedColIndC,
void*                    pBuffer)
```

```c
cusparseStatus_t
cusparseDcsrgem2(cusparseHandle_t         handle,
int                      m,
int                      n,
const double*            alpha,
const cusparseMatDescr_t descrA,
int                      nnzA,
const double*            csrSortedValA,
const int*               csrSortedRowPtrA,
const int*               csrSortedColIndA,
const double*            beta,
const cusparseMatDescr_t descrB,
int                      nnzB,
const double*            csrSortedValB,
const int*               csrSortedRowPtrB,
const int*               csrSortedColIndB,
const cusparseMatDescr_t descrC,
double*                  csrSortedValC,
int*                     csrSortedRowPtrC,
int*                     csrSortedColIndC,
void*                    pBuffer)
```

```c
cusparseStatus_t
cusparseCcsrgem2(cusparseHandle_t         handle,
int                      m,
int                      n,
const cuComplex*         alpha,
const cusparseMatDescr_t descrA,
int                      nnzA,
const cuComplex*         csrSortedValA,
const int*               csrSortedRowPtrA,
const int*               csrSortedColIndA,
```
This function performs following matrix-matrix operation

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

where A, B, and C are $m \times n$ sparse matrices (defined in CSR storage format by the three arrays \( \text{csrValA} | \text{csrValB} | \text{csrValC}, \text{csrRowPtrA} | \text{csrRowPtrB} | \text{csrRowPtrC}, \text{and csrColIndA} | \text{csrColIndB} | \text{csrColIndC} \) respectively), and $\alpha$ and $\beta$ are scalars. Since A and B have different sparsity patterns, cuSPARSE adopts a two-step approach to complete sparse matrix C. In the first step, the user allocates $\text{csrRowPtrC}$ of $m+1$ elements and uses function \text{cusparseXcsrgeam2Nnz()} to determine $\text{csrRowPtrC}$ and the total number of nonzero elements. In the second step, the user gathers $\text{nnzC}$ (number of nonzero elements of matrix C) from either ($\text{nnzC} = *\text{nnzTotalDevHostPtr}$) or ($\text{nnzC} = \text{csrRowPtrC}(m) - \text{csrRowPtrC}(0)$) and allocates $\text{csrValC}, \text{csrColIndC}$ of $\text{nnzC}$ elements respectively, then finally calls function \text{cusparse[S|D|C|Z]csrgeam2()} to complete matrix C.

The general procedure is as follows:

```c
int baseC, nnzC;
/* alpha, nnzTotalDevHostPtr points to host memory */
size_t BufferSizeInBytes;
char *buffer = NULL;
int *nnzTotalDevHostPtr = &nnzC;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
/* prepare buffer */
cusparseScsrgeam2_bufferSizeExt(handle, m, n,
   alpha,
   const cuComplex* beta,
   const cusparseMatDescr_t descrB,
   int nnzB,
   const cuComplex* csrSortedValB,
   const int* csrSortedRowPtrB,
   const int* csrSortedColIndB,
   const cusparseMatDescr_t descrC,
   int* csrSortedValC,
   int* csrSortedRowPtrC,
   int* csrSortedColIndC,
   void* pBuffer)
```
Several comments on `csrgeam2()`:

- The other three combinations, NT, TN, and TT, are not supported by cuSPARSE. In order to do any one of the three, the user should use the routine `csr2csc()` to convert $A|B$ to $A^T|B^T$.

- Only `CUSPARSE_MATRIX_TYPE_GENERAL` is supported. If either $A$ or $B$ is symmetric or Hermitian, then the user must extend the matrix to a full one and reconfigure the `MatrixType` field of the descriptor to `CUSPARSE_MATRIX_TYPE_GENERAL`.

- If the sparsity pattern of matrix $C$ is known, the user can skip the call to function `cusparseXcsrgeam2Nnz()`. For example, suppose that the user has an iterative algorithm which would update $A$ and $B$ iteratively but keep the sparsity patterns. The user can call function `cusparseXcsrgeam2Nnz()` once to set up the sparsity pattern of $C$, then call function `cusparse[S|D|C|Z]geam()` only for each iteration.

- The pointers `alpha` and `beta` must be valid.

- When `alpha` or `beta` is zero, it is not considered a special case by cuSPARSE. The sparsity pattern of $C$ is independent of the value of `alpha` and `beta`. If the user wants $C = 0 \times A + 1 \times B^T$, then `csr2csc()` is better than `csrgeam2()`.

- `csrgeam2()` is the same as `csrgeam()` except `csrgeam2()` needs explicit buffer where `csrgeam()` allocates the buffer internally.

- This function requires temporary extra storage that is allocated internally.

- The routine does not support asynchronous execution.
The routine does not support CUDA graph capture.

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of sparse matrix A, B, C.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of sparse matrix A, B, C.</td>
</tr>
<tr>
<td>alpha</td>
<td>&lt;type&gt; scalar used for multiplication.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL only.</td>
</tr>
<tr>
<td>nnzA</td>
<td>number of nonzero elements of sparse matrix A.</td>
</tr>
<tr>
<td>csrValA</td>
<td>&lt;type&gt; array of nnzA(= csrRowPtrA(m) - csrRowPtrA(0)) nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of m + 1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of nnzA(= csrRowPtrA(m) - csrRowPtrA(0)) column indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td>beta</td>
<td>&lt;type&gt; scalar used for multiplication. If beta is zero, y does not have to be a valid input.</td>
</tr>
<tr>
<td>descrB</td>
<td>the descriptor of matrix B. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL only.</td>
</tr>
<tr>
<td>nnzB</td>
<td>number of nonzero elements of sparse matrix B.</td>
</tr>
<tr>
<td>csrValB</td>
<td>&lt;type&gt; array of nnzB(= csrRowPtrB(m) - csrRowPtrB(0)) nonzero elements of matrix B.</td>
</tr>
<tr>
<td>csrRowPtrB</td>
<td>integer array of m + 1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndB</td>
<td>integer array of nnzB(= csrRowPtrB(m) - csrRowPtrB(0)) column indices of the nonzero elements of matrix B.</td>
</tr>
<tr>
<td>descrC</td>
<td>the descriptor of matrix C. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL only.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValC</td>
<td>&lt;type&gt; array of nnzC(= csrRowPtrC(m) - csrRowPtrC(0)) nonzero elements of matrix C.</td>
</tr>
<tr>
<td>csrRowPtrC</td>
<td>integer array of m + 1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>integer array of nnzC(= csrRowPtrC(m) - csrRowPtrC(0)) column indices of the nonzero elements of matrix C.</td>
</tr>
</tbody>
</table>
### nnzTotalDevHostPtr

The total number of nonzero elements in device or host memory. It is equal to \((\text{csrRowPtrC}(m) - \text{csrRowPtrC}(0))\).

See `cusparseStatus_t` for the description of the return status.

### 10.2. `cusparse<t>csrgemm2()` [DEPRECATED]

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

```c
cusparseStatus_t cusparseScsrgemm2_bufferSizeExt(cusparseHandle_t handle, int m, int n, int k, const float* alpha, const cusparseMatDescr_t descrA, int nnzA, const int* csrRowPtrA, const int* csrColIndA, const cusparseMatDescr_t descrB, int nnzB, const int* csrRowPtrB, const int* csrColIndB, const float* beta, const cusparseMatDescr_t descrD, int nnzD, const int* csrRowPtrD, const int* csrColIndD, csrgemm2Info_t info, size_t* pBufferSizeInBytes)

cusparseStatus_t cusparseDcsrgemm2_bufferSizeExt(cusparseHandle_t handle, int m, int n, int k, const double* alpha, const cusparseMatDescr_t descrA, int nnzA, const int* csrRowPtrA, const int* csrColIndA, const cusparseMatDescr_t descrB, int nnzB, const int* csrRowPtrB, const int* csrColIndB, const double* beta, const cusparseMatDescr_t descrD, int nnzD, const int* csrRowPtrD, const int* csrColIndD, csrgemm2Info_t info, size_t* pBufferSizeInBytes)
```
cusparseStatus_t
cusparseCcsrgemm2_bufferSizeExt(cusparseHandle_t handle, int m, int n, int k, const cuComplex* alpha, const cusparseMatDescr_t descrA, int nnzA, const int* csrRowPtrA, const int* csrColIndA, const cusparseMatDescr_t descrB, int nnzB, const int* csrRowPtrB, const int* csrColIndB, const cuComplex* beta, const cusparseMatDescr_t descrC, int nnzC, const int* csrRowPtrC, const int* csrColIndC, csrgemm2Info_t info, size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsrgemm2_bufferSizeExt(cusparseHandle_t handle, int m, int n, int k, const cuDoubleComplex* alpha, const cusparseMatDescr_t descrA, int nnzA, const int* csrRowPtrA, const int* csrColIndA, const cusparseMatDescr_t descrB, int nnzB, const int* csrRowPtrB, const int* csrColIndB, const cuDoubleComplex* beta, const cusparseMatDescr_t descrC, int nnzC, const int* csrRowPtrC, const int* csrColIndC, csrgemm2Info_t info, size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseXcsrgemm2Nnz(cusparseHandle_t handle, int m, int n, int k, 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 nnzC, const int* csrRowPtrC, const int* csrColIndC, int nnzD, const int* csrRowPtrD, const int* csrColIndD, int nnzE, const int* csrRowPtrE, const int* csrColIndE)
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
const csrgemm2Info_t info,
void* pBuffer)

 cusparseStatus_t
cusparseScsrgemm2(cusparseHandle_t handle,
int m,
int n,
int k,
const float* alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const cusparseMatDescr_t descrB,
int nnzB,
const float* csrValB,
const int* csrRowPtrB,
const int* csrColIndB,
const float* beta,
const cusparseMatDescr_t descrD,
int nnzD,
const float* csrValD,
const int* csrRowPtrD,
const int* csrColIndD,
const cusparseMatDescr_t descrC,
float* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
const csrgemm2Info_t info,
void* pBuffer)

cusparseStatus_t
cusparseDcsrgemm2(cusparseHandle_t handle,
int m,
int n,
int k,
const double* alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const cusparseMatDescr_t descrB,
int nnzB,
const double* csrValB,
const int* csrRowPtrB,
const int* csrColIndB,
const double* beta,
const cusparseMatDescr_t descrD,
int nnzD,
const double* csrValD,
const int* csrRowPtrD,
const int* csrColIndD,
const cusparseMatDescr_t descrC,
double* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
const csrgemm2Info_t     info,
void*                    pBuffer)

cusparseStatus_t
cusparseCcsrgemm2(cusparseHandle_t handle,
int m,
int n,
int k,
const cuComplex*        alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const cuComplex*        csrValA,
const int*              csrRowPtrA,
const int*              csrColIndA,
const cusparseMatDescr_t descrB,
int nnzB,
const cuComplex*        csrValB,
const int*              csrRowPtrB,
const int*              csrColIndB,
const cuComplex*        beta,
const cusparseMatDescr_t descrC,
int* csrColIndC,
const cuComplex*        csrValC,
const int*              csrRowPtrC,
const int*              csrRowPtrD,
const cusparseMatDescr_t descrD,
int nnzD,
const cuComplex*        csrValD,
const int*              csrRowPtrD,
const int*              csrColIndD,
const cuComplex*        csrValC,
const int*              csrRowPtrC,
const int*              csrColIndC,
const csrgemm2Info_t     info,
void*                    pBuffer)

cusparseStatus_t
cusparseZcsrgemm2(cusparseHandle_t handle,
int m,
int n,
int k,
const cuDoubleComplex*   alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const cuDoubleComplex*   csrValA,
const int*              csrRowPtrA,
const int*              csrColIndA,
const cusparseMatDescr_t descrB,
int nnzB,
const cuDoubleComplex*   csrValB,
const int*              csrRowPtrB,
const int*              csrColIndB,
const cuDoubleComplex*   beta,
const cusparseMatDescr_t descrC,
int* csrColIndC,
const cuDoubleComplex*   csrValC,
const int*              csrRowPtrC,
const int*              csrRowPtrD,
const cusparseMatDescr_t descrD,
int nnzD,
const cuDoubleComplex*   csrValD,
const int*              csrRowPtrD,
const int*              csrColIndD,
const cuDoubleComplex*   csrValC,
const int*              csrRowPtrC,
const int*              csrColIndC,
const csrgemm2Info_t     info,
void*                    pBuffer)
This function performs following matrix-matrix operation:

\[ C = \alpha A \ast B + \beta D \]

where \( A, B, D \) and \( C \) are \( m \times k, k \times n, m \times n \) and \( m \times n \) sparse matrices (defined in CSR storage format by the three arrays \( \text{csrValA|csrValB|csrValD|csrValC, csrRowPtrA|csrRowPtrB|csrRowPtrD|csrRowPtrC}, \)
\( \text{csrColIndA|csrColIndB|csrColIndD|csrColIndC} \) respectively.

Note that the new API \( \text{cusparseSpGEMM} \) requires that \( D \) must have the same sparsity pattern of \( C \).

The \( \text{csrgemm2} \) uses \( \alpha \) and \( \beta \) to support the following operations:

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \beta )</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>invalid</td>
</tr>
<tr>
<td>NULL</td>
<td>!NULL</td>
<td>( C = \beta D ), A and B are not used</td>
</tr>
<tr>
<td>!NULL</td>
<td>NULL</td>
<td>( C = \alpha AB ), D is not used</td>
</tr>
<tr>
<td>!NULL</td>
<td>!NULL</td>
<td>( C = \alpha AB + \beta D )</td>
</tr>
</tbody>
</table>

The numerical value of \( \alpha \) and \( \beta \) only affects the numerical values of \( C \), not its sparsity pattern. For example, if \( \alpha \) and \( \beta \) are not zero, the sparsity pattern of \( C \) is union of \( A \ast B \) and \( D \), independent of numerical value of \( \alpha \) and \( \beta \).

The following table shows different operations according to the value of \( m, n \) and \( k \):

<table>
<thead>
<tr>
<th>( m, n, k )</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m&lt;0 ) or ( n&lt;0 ) or ( k&lt;0 )</td>
<td>invalid</td>
</tr>
<tr>
<td>( m=0 ) or ( n=0 )</td>
<td>do nothing</td>
</tr>
<tr>
<td>( m&gt;0 ) and ( n&gt;0 ) and ( k=0 )</td>
<td>invalid if ( \beta ) is zero;\n( C = \beta D ) if ( \beta ) is not zero.</td>
</tr>
<tr>
<td>( m&gt;0 ) and ( n&gt;0 ) and ( k&gt;0 )</td>
<td>( C = \beta D ) if ( \alpha ) is zero.\n( C = \alpha AB ) if ( \alpha ) is zero.\n( C = \alpha AB + \beta D ) if ( \alpha ) and ( \beta ) are not zero.</td>
</tr>
</tbody>
</table>

This function requires the buffer size returned by \( \text{csrgemm2_bufferSizeExt()} \). The address of \( \text{pBuffer} \) must be multiple of 128 bytes. If it is not, \( \text{CUSPARSE_STATUS_INVALID_VALUE} \) is returned.

The cuSPARSE library adopts a two-step approach to complete sparse matrix. In the first step, the user allocates \( \text{csrRowPtrC} \) of \( m+1 \) elements and uses the function \( \text{cusparseXcsrgemm2Nnz()} \) to determine \( \text{csrRowPtrC} \) and the total number of nonzero elements. In the second step, the user gathers \( \text{nnzC} \) (the number of nonzero elements of matrix \( C \)) from either \( \text{(nnzC=nnzTotalDevHostPtr)} \) or \( \text{(nnzC=csrRowPtrC(m)-csrRowPtrC(0))} \) and allocates \( \text{csrValC} \) and \( \text{csrColIndC} \) of \( \text{nnzC} \) elements respectively, then finally calls function \( \text{cusparse[S|D|C|Z]csrgemm2()} \) to evaluate matrix \( C \).

The general procedure of \( C=-A \ast B+D \) is as follows:

```c
// assume matrices A, B and D are ready.
```
int baseC, nnzC;
csrmm2Info_t info = NULL;
size_t bufferSize;
void *buffer = NULL;
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzC;
double alpha = -1.0;
double beta = 1.0;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);

// step 1: create an opaque structure
cusparseCreateCsrgemm2Info(&info);

// step 2: allocate buffer for csrgemm2Nnz and csrgemm2

cusparseDcsrgemm2_bufferSizeExt(handle, m, n, k, &alpha,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    &beta,
    descrC, csrRowPtrC, csrColIndC,
    info,
    &bufferSize);
cudaMalloc(&buffer, bufferSize);

// step 3: compute csrRowPtrC
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgemm2Nnz(handle, m, n, k,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    descrC, csrRowPtrC, nnzTotalDevHostPtr,
    info, buffer);
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
} else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}

// step 4: finish sparsity pattern and value of C

cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(double)*nnzC);
// Remark: set csrValC to null if only sparsity pattern is required.
cusparseDcsrgemm2(handle, m, n, k, &alpha,
    descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrValB, csrRowPtrB, csrColIndB,
    &beta,
    descrC, csrValC, csrRowPtrC, csrColIndC,
    info, buffer);

// step 5: destroy the opaque structure
cusparseDestroyCsrgemm2Info(info);

Several comments on csrgemm2():

- Only the NN version is supported. For other modes, the user has to transpose A or B explicitly.
- Only CUSPARSE_MATRIX_TYPE_GENERAL is supported. If either A or B is symmetric or Hermitian, the user must extend the matrix to a full one and reconfigure the MatrixType field descriptor to CUSPARSE_MATRIX_TYPE_GENERAL.
- If csrValC is zero, only sparsity pattern of C is calculated.
The functions `cusparseXcsrgem2Nnz()` and `cusparse<t>csrgem2()` support the following properties if `pBuffer != NULL`:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine does **not** support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows of sparse matrix A, D and C.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>number of columns of sparse matrix B, D and C.</td>
</tr>
<tr>
<td><code>k</code></td>
<td>number of columns/rows of sparse matrix A / B.</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td><code>&lt;type&gt;</code> scalar used for multiplication.</td>
</tr>
<tr>
<td><code>descrA</code></td>
<td>the descriptor of matrix A. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code> only.</td>
</tr>
<tr>
<td><code>nnzA</code></td>
<td>number of nonzero elements of sparse matrix A.</td>
</tr>
<tr>
<td><code>csrValA</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnzA</code> nonzero elements of matrix A.</td>
</tr>
<tr>
<td><code>csrRowPtrA</code></td>
<td>integer array of <code>m+1</code> elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td><code>csrColIndA</code></td>
<td>integer array of <code>nnzA</code> column indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td><code>descrB</code></td>
<td>the descriptor of matrix B. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code> only.</td>
</tr>
<tr>
<td><code>nnzB</code></td>
<td>number of nonzero elements of sparse matrix B.</td>
</tr>
<tr>
<td><code>csrValB</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnzB</code> nonzero elements of matrix B.</td>
</tr>
<tr>
<td><code>csrRowPtrB</code></td>
<td>integer array of <code>k+1</code> elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td><code>csrColIndB</code></td>
<td>integer array of <code>nnzB</code> column indices of the nonzero elements of matrix B.</td>
</tr>
<tr>
<td><code>descrD</code></td>
<td>the descriptor of matrix D. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code> only.</td>
</tr>
<tr>
<td><code>nnzD</code></td>
<td>number of nonzero elements of sparse matrix D.</td>
</tr>
<tr>
<td><code>csrValD</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnzD</code> nonzero elements of matrix D.</td>
</tr>
<tr>
<td><code>csrRowPtrD</code></td>
<td>integer array of <code>m+1</code> elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td><code>csrColIndD</code></td>
<td>integer array of <code>nnzD</code> column indices of the nonzero elements of matrix D.</td>
</tr>
<tr>
<td><code>beta</code></td>
<td><code>&lt;type&gt;</code> scalar used for multiplication.</td>
</tr>
<tr>
<td>descrC</td>
<td>the descriptor of matrix C. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL only.</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>info</td>
<td>structure with information used in csrgemm2Nnz and csrgemm2.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user; the size is returned by csrgemm2_bufferSizeExt.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>csrValC</th>
<th>&lt;type&gt; array of nnzC nonzero elements of matrix C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrRowPtrC</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>integer array of nnzC column indices of the nonzero elements of matrix C.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>number of bytes of the buffer used in csrgemm2Nnz and csrgemm2.</td>
</tr>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>total number of nonzero elements in device or host memory. It is equal to (csrRowPtrC(m) - csrRowPtrC(0)).</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.
Chapter 11. cuSPARSE Preconditioners Reference

This chapter describes the routines that implement different preconditioners.

11.1. Incomplete Cholesky Factorization: level 0

Different algorithms for ic0 are discussed in this section.

11.1.1. cusparse<t>csric02_bufferSize()

```c
 cusparseStatus_t
 cusparseScsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            float*                   csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)
```

```c
 cusparseStatus_t
 cusparseDcsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            double*                  csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)
```

```c
 cusparseStatus_t
 cusparseCcsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            cuComplex*               csrValA,
                            const int*               csrRowPtrA,
```

```c
                           int*                     pBufferSizeInBytes)
```
This function returns size of buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

\[ A \approx LL^H \]

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

The buffer size depends on dimension \( m \) and \( \text{nnz} \), the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call \( \text{csric02_bufferSize()} \) again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( m )</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{nnz} )</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>( \text{csrValA} )</td>
<td>( &lt;\text{type}&gt; ) array of ( \text{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{csrRowPtrA} )</td>
<td>integer array of ( m + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>( \text{csrColIndA} )</td>
<td>integer array of ( \text{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>

**Output**
### 11.1.2. `cusparse<t>csric02_analysis()`

<table>
<thead>
<tr>
<th>info</th>
<th>record internal states based on different algorithms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pBufferSizeInBytes</td>
<td>number of bytes of the buffer used in <code>csric02_analysis()</code> and <code>csric02()</code>.</td>
</tr>
</tbody>
</table>

See **`cusparseStatus_t`** for the description of the return status.
This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

\[ A = LL^T \]

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

This function requires a buffer size returned by \texttt{csric02_bufferSize()}. The address of pBuffer must be multiple of 128 bytes. If not, \texttt{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

Function \texttt{csric02\_analysis()} reports a structural zero and computes level information stored in the opaque structure info. The level information can extract more parallelism during incomplete Cholesky factorization. However \texttt{csric02()} can be done without level information. To disable level information, the user must specify the policy of \texttt{csric02\_analysis()} and \texttt{csric02()} as \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}.

Function \texttt{csric02\_analysis()} always reports the first structural zero, even if the policy is \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}. The user needs to call \texttt{cusparseXcsric02\_zeroPivot()} to know where the structural zero is.

It is the user’s choice whether to call \texttt{csric02()} if \texttt{csric02\_analysis()} reports a structural zero. In this case, the user can still call \texttt{csric02()}, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

\textbf{Input}

<table>
<thead>
<tr>
<th>\texttt{handle}</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m}</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{nnz}</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{descrA}</td>
<td>the descriptor of matrix ( A ). The supported matrix type is \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are \texttt{CUSPARSE_INDEX_BASE_ZERO} and \texttt{CUSPARSE_INDEX_BASE_ONE}.</td>
</tr>
<tr>
<td>\texttt{csrValA}</td>
<td>&lt;type&gt; array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{csrRowPtrA}</td>
<td>integer array of ( m + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>\texttt{csrColIndA}</td>
<td>integer array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>\texttt{info}</td>
<td>structure initialized using \texttt{cusparseCreateCsric02Info()}.</td>
</tr>
</tbody>
</table>
The supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.

Buffer allocated by the user; the size is returned by `csric02_bufferSize()`.

Number of bytes of the buffer used in `csric02_analysis()` and `csric02()`.

See `cusparseStatus_t` for the description of the return status.

### 11.1.3. `cusparse<t>csric02()`

```c
cusparseStatus_t
cusparseScsric02(cusparseHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
float* csrValA_valM,
const int* csrRowPtrA,
const int* csrColIndA,
csric02Info_t info,
cusparseSolvePolicy_t policy,
void* pBuffer)

cusparseStatus_t
cusparseDcsric02(cusparseHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
double* csrValA_valM,
const int* csrRowPtrA,
const int* csrColIndA,
info,
cusparseSolvePolicy_t policy,
pBuffer)

cusparseStatus_t
cusparseCcsric02(cusparseHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
cuComplex* csrValA_valM,
const int* csrRowPtrA,
const int* csrColIndA,
info,
cusparseSolvePolicy_t policy,
pBuffer)

cusparseStatus_t
cusparseZcsric02(cusparseHandle_t handle,
int m,
int nnz,
const cusparseMatDescr_t descrA,
cuDoubleComplex* csrValA_valM,
const int* csrRowPtrA,
```
This function performs the solve phase of the computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

\[ A \approx LL^H \]

This function requires a buffer size returned by `csric02_bufferSize()`. The address of pBuffer must be a multiple of 128 bytes. If not, `CUSPARSE_STATUS_INVALID_VALUE` is returned.

Although `csric02()` can be done without level information, the user still needs to be aware of consistency. If `csric02_analysis()` is called with policy `CUSPARSE_SOLVE_POLICY_USE_LEVEL`, `csric02()` can be run with or without levels. On the other hand, if `csric02_analysis()` is called with `CUSPARSE_SOLVE_POLICY_NO_LEVEL`, `csric02()` can only accept `CUSPARSE_SOLVE_POLICY_NO_LEVEL`; otherwise, `CUSPARSE_STATUS_INVALID_VALUE` is returned.

Function `csric02()` reports the first numerical zero, including a structural zero. The user must call `cusparseXcsric02_zeroPivot()` to know where the numerical zero is.

Function `csric02()` only takes the lower triangular part of matrix \( A \) to perform factorization. The matrix type must be `CUSPARSE_MATRIX_TYPE_GENERAL`, the fill mode and diagonal type are ignored, and the strictly upper triangular part is ignored and never touched. It does not matter if \( A \) is Hermitian or not. In other words, from the point of view of `csric02()` \( A \) is Hermitian and only the lower triangular part is provided.

**Note:** In practice, a positive definite matrix may not have incomplete cholesky factorization. To the best of our knowledge, only matrix \( M \) can guarantee the existence of incomplete cholesky factorization. If `csric02()` failed cholesky factorization and reported a numerical zero, it is possible that incomplete cholesky factorization does not exist.

For example, suppose \( A \) is a real \( m \times m \) matrix, the following code solves the precondition system \( M \mathbf{y} = \mathbf{x} \) where \( M \) is the product of Cholesky factorization \( L \) and its transpose.

\[ M = LL^H \]

```c
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory,
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
csric02Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
```
```c
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal

cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create an empty info structure
// we need one info for csric02 and two info's for csrsv2

cusparseCreateCsric02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_Lt);

// step 3: query how much memory used in csric02 and csrsv2, and allocate the buffer

cusparseDcsric02_bufferSize(handle, m, nnz,
    descr_M, d csrVal, d csrRowPtr, d csrColInd, info_M, pBufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
    descr_L, d csrVal, d csrRowPtr, d csrColInd, info_L, pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_Lt, m, nnz,
    descr_L, d csrVal, d csrRowPtr, d csrColInd, info_Lt, pBufferSize_Lt);

pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

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

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// we can do analysis of csric02 and csrsv2 simultaneously.

cusparseDcsric02_analysis(handle, m, nnz, descr_M,
    d csrVal, d csrRowPtr, d csrColInd, info_M, policy_M, pBuffer);

status = cusparseXcsric02_zeroPivot(handle, info_M, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
    printf("A(%d,%d) is missing
", structural_zero, structural_zero);
}

cusparseDcsrsv2_analysis(handle, trans_L, m, nnz, descr_L,
    d csrVal, d csrRowPtr, d csrColInd, info_L, policy_L, pBuffer);

// step 5: M = L * L'
cusparseDcsric02(handle, m, nnz, descr_M,
```
The function supports the following properties if pBuffer != NULL

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzeros of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA_valM</td>
<td>&lt;type&gt; array of nnz((= csrRowPtrA(m) - csrRowPtrA(0 )) nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of m + 1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of nnz((= csrRowPtrA(m) - csrRowPtrA(0 )) column indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td>info</td>
<td>structure with information collected during the analysis phase [that should have been passed to the solve phase unchanged].</td>
</tr>
<tr>
<td>policy</td>
<td>the supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.</td>
</tr>
</tbody>
</table>
**11.1.4. **`cusparseXcsric02_zeroPivot()`

```c
 cusparseStatus_t
cusparseXcsric02_zeroPivot(cusparseHandle_t handle,
                          csric02Info_t info,
                          int* position)
```

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

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

Function `cusparseXcsric02_zeroPivot()` is a blocking call. It calls `cudaDeviceSynchronize()` to make sure all previous kernels are done.

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

- The routine requires no extra storage
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
</table>
| info            | info contains structural zero or numerical zero if the user already called `csric02_analysis()` or `csric02()`.

**Output**

| position        | if no structural or numerical zero, `position` is `-1`; otherwise, if $A(j,j)$ is missing or $L(j,j)$ is zero, `position=j`. |

See `cusparseStatus_t` for the description of the return status.

**11.1.5. **`cusparse<t>bsric02_bufferSize()`

```c
 cusparseStatus_t
cusparseSbsric02_bufferSize(cusparseHandle_t handle,
                          cusparseDirection_t dirA,
                          int mb,)
```

See `cusparseStatus_t` for the description of the return status.
This function returns the size of a buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting

\[ A \approx LL^H \]

\( A \) is an \((mb \times blockDim) \times (mb \times blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays \( \text{bsrValA} \), \( \text{bsrRowPtrA} \), and \( \text{bsrColIndA} \).

The buffer size depends on the dimensions of \( mb \), \( blockDim \), and the number of nonzero blocks of the matrix \( nnzb \). If the user changes the matrix, it is necessary to call \( \text{bsric02_bufferSize()} \) again to have the correct buffer size; otherwise, a segmentation fault may occur.
The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirA</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows and block columns of matrix A.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; array of ( \text{nnzb} = \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0) ) nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of ( \text{mb} + 1 ) elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of ( \text{nnzb} = \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0) ) column indices of the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A, larger than zero.</td>
</tr>
</tbody>
</table>

**Output**

| info       | record internal states based on different algorithms. |
| pBufferSizeInBytes | number of bytes of the buffer used in bsric02_analysis() and bsric02(). |

See **cusparseStatus_t** for the description of the return status

**11.1.6. cusparse<t>bsric02_analysis()**

```c
 cusparseStatus_t
cusparseSbsric02_analysis(cusparseHandle_t handle,          
cusparseDirection_t dirA,                      
int mb,                                        
int nnzb,                                      
cusparseMatDescr_t descrA,                    
const float* bsrValA,                         
const int* bsrRowPtrA,                        
const int* bsrColIndA,                        
int blockDim,                                 
```

```c
 cusparseStatus_t
cusparseDbsric02_analysis(cusparseHandle_t handle,          
cusparseDirection_t dirA,                      
int mb,                                        
int nnzb,                                      
cusparseMatDescr_t descrA,                    
const float* bsrValA,                         
const int* bsrRowPtrA,                        
const int* bsrColIndA,                        
int blockDim,                                 
```
This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

\[ A \approx L L^H \]

\( A \) is an \((mb*blockDim) \times (mb*blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays  \(\text{bsrValA}, \text{bsrRowPtrA}, \) and \(\text{bsrColIndA} \). The block in BSR format is of size \(blockDim*blockDim\), stored as column-major or row-major as determined by parameter  \(\text{dirA}\), which is either \text{CUSPARSE_DIRECTION_COLUMN} or \text{CUSPARSE_DIRECTION_ROW}. The matrix type must be \text{CUSPARSE_MATRIX_TYPE_GENERAL}, and the fill mode and diagonal type are ignored.
This function requires a buffer size returned by `bsric02_bufferSize90`. The address of `pBuffer` must be a multiple of 128 bytes. If it is not, `CUSPARSE_STATUS_INVALID_VALUE` is returned.

Function `bsric02_analysis()` reports structural zero and computes level information stored in the opaque structure `info`. The level information can extract more parallelism during incomplete Cholesky factorization. However `bsric02()` can be done without level information. To disable level information, the user needs to specify the parameter `policy` of `bsric02[_[analysis|] ] as CUSPARSE_SOLVE_POLICY_NO_LEVEL.

Function `bsric02_analysis` always reports the first structural zero, even when parameter `policy` is `CUSPARSE_SOLVE_POLICY_NO_LEVEL`. The user must call `cusparseXbsric02_zeroPivot()` to know where the structural zero is.

It is the user’s choice whether to call `bsric02()` if `bsric02_analysis()` reports a structural zero. In this case, the user can still call `bsric02()`, which returns a numerical zero in the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or \</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows and block columns of matrix A.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_INDEX_BASE_ZERO and \</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>bsrValA</td>
<td><code>&lt;type&gt;</code> array of ( nnzb( = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0) ) ) nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of ( mb + 1 ) elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of ( nnzb( = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0) ) ) column indices of the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A; must be larger than zero.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using <code>cusparseCreateBsric02Info()</code></td>
</tr>
</tbody>
</table>
The supported policies are 
CUSPARSE_SOLVE_POLICY_NO_LEVEL and 
CUSPARSE_SOLVE_POLICY_USE_LEVEL.

buffer allocated by the user; the size is returned 
by bsric02_bufferSize().

structure filled with information collected during 
the analysis phase [that should be passed to the 
solve phase unchanged].

See cusparseStatus_t for the description of the return status

### 11.1.7. cusparse<t>bsric02()
This function performs the solve phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

\[ A \approx LL^H \]

A is an \((mb\times blockDim)\times(mb\times blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays bsrValA, bsrRowPtrA, and bsrColIndA. The block in BSR format is of size blockDim\times blockDim, stored as column-major or row-major as determined by parameter dirA, which is either CUSPARSE_DIRECTION_COLUMN or CUSPARSE_DIRECTION_ROW. The matrix type must be CUSPARSE_MATRIX_TYPE_GENERAL, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by bsric02_bufferSize(). The address of pBuffer must be a multiple of 128 bytes. If it is not, CUSPARSE_STATUS_INVALID_VALUE is returned.

Although bsric02() can be done without level information, the user must be aware of consistency. If bsric02_analysis() is called with policy CUSPARSE_SOLVE_POLICY_USE_LEVEL, bsric02() can be run with or without levels. On the other hand, if bsric02_analysis() is called with CUSPARSE_SOLVE_POLICY_NO_LEVEL, bsric02() can only accept CUSPARSE_SOLVE_POLICY_NO_LEVEL; otherwise, CUSPARSE_STATUS_INVALID_VALUE is returned.

Function bsric02() has the same behavior as csric02(). That is, bsr2csr(bsric02(A)) = csric02(bsr2csr(A)). The numerical zero of csric02() means there exists some zero \( L(j,j) \). The numerical zero of bsric02() means there exists some block \( L(j,j) \) that is not invertible.

Function bsric02 reports the first numerical zero, including a structural zero. The user must call cusparseXbsric02_zeroPivot() to know where the numerical zero is.

The bsric02() function only takes the lower triangular part of matrix A to perform factorization. The strictly upper triangular part is ignored and never touched. It does not matter if A is Hermitian or not. In other words, from the point of view of bsric02(), A is Hermitian and only the lower triangular part is provided. Moreover, the imaginary part of diagonal elements of diagonal blocks is ignored.

For example, suppose A is a real m-by-m matrix, where \( m=mb\times blockDim \). The following code solves precondition system \( M^*y = x \), where M is the product of Cholesky factorization L and its transpose.
\[ M = LL^H \]

```c
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
bsric02Info_t info_M  = 0;
bsrsv2Info_t  info_L  = 0;
bsrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsric02 and two info's for bsrsv2
cusparseCreateBsric02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_Lt);

// step 3: query how much memory used in bsric02 and bsrsv2, and allocate the buffer
cusparseDbsric02_bufferSize(handle, dir, mb, nnzb,
    descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &bufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &bufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_Lt, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &bufferSize_Lt);
pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));
// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
```
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
//        perform analysis of triangular solve on L
//        perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// we can do analysis of bsric02 and bsrsv2 simultaneously.

cusparseDbsric02_analysis(handle, dir, mb, nnzb, descr_M,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim, info_M,
   policy_M, pBuffer);
status = cusparseXbsric02_zeroPivot(handle, info_M, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("A(%d,%d) is missing\n", structural_zero, structural_zero);
}
cusparseDbsrsv2_analysis(handle, dir, trans_L, mb, nnzb, descr_L,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim,
   info_L, policy_L, pBuffer);
cusparseDbsrsv2_analysis(handle, dir, trans_Lt, mb, nnzb, descr_L,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim,
   info_Lt, policy_Lt, pBuffer);

// step 5: M = L * L'
cusparseDbsric02_solve(handle, dir, mb, nnzb, descr_M,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim, info_M, policy_M, pBuffer);
status = cusparseXbsric02_zeroPivot(handle, info_M, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is not positive definite\n", numerical_zero, numerical_zero);
}

// step 6: solve L'*z = x
cusparseDbsrsv2_solve(handle, dir, trans_L, mb, nnzb, &alpha, descr_L,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim, info_L,
   d_x, d_z, policy_L, pBuffer);

// step 7: solve L'*y = z
cusparseDbsrsv2_solve(handle, dir, trans_Lt, mb, nnzb, &alpha, descr_L,
   &d_bsrVal, &d_bsrRowPtr, &d_bsrColInd, blockDim, info_Lt,
   d_z, d_y, policy_Lt, pBuffer);

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyMatDescr(descr_M);
cusparseDestroyMatDescr(descr_L);
cusparseDestroyBsric02Info(info_M);
cusparseDestroyBsrsv2Info(info_L);
cusparseDestroyBsrsv2Info(info_Lt);
cusparseDestroy(handle);

The function supports the following properties if pBuffer != NULL

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirA</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
</tbody>
</table>
### mb
number of block rows and block columns of matrix \(A\).

### nnzb
number of nonzero blocks of matrix \(A\).

### descrA
the descriptor of matrix \(A\). The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}.
Also, the supported index bases are \texttt{CUSPARSE\_INDEX\_BASE\_ZERO} and \texttt{CUSPARSE\_INDEX\_BASE\_ONE}.

### bsrValA
\(<\text{type}>\) array of \(\text{nnzb} (= \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0))\) nonzero blocks of matrix \(A\).

### bsrRowPtrA
integer array of \(\text{mb} + 1\) elements that contains the start of every block row and the end of the last block row plus one.

### bsrColIndA
integer array of \(\text{nnzb} (= \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0))\) column indices of the nonzero blocks of matrix \(A\).

### blockDim
block dimension of sparse matrix \(A\), larger than zero.

### info
structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).

### policy
the supported policies are \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL} and \texttt{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL}.

### pBuffer
buffer allocated by the user, the size is returned by \texttt{bsric02_bufferSize()}.

### Output

| bsrValA | \(<\text{type}>\) matrix containing the incomplete-Cholesky lower triangular factor. |

See \texttt{cusparseStatus\_t} for the description of the return status.

### 11.1.8. cusparseXbsric02_zeroPivot()

```c

cusparseStatus_t
cusparseXbsric02_zeroPivot(cusparseHandle_t handle, 
                          bsric02Info_t info, 
                          int* position)
```

If the returned error code is \texttt{CUSPARSE\_STATUS\_ZERO\_PIVOT}, \(\text{position}=j\) means \(A(j,j)\) has either a structural zero or a numerical zero (the block is not positive definite). Otherwise \(\text{position}=-1\).

The \texttt{position} can be 0-based or 1-based, the same as the matrix.

Function \texttt{cusparseXbsric02_zeroPivot()} is a blocking call. It calls \texttt{cudaDeviceSynchronize()} to make sure all previous kernels are done.
The position can be in the host memory or device memory. The user can set the proper mode with cusparseSetPointerMode().

- The routine requires no extra storage
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>info contains a structural zero or a numerical zero if the user already called bsric02_analysis() or bsric02().</td>
</tr>
</tbody>
</table>

**Output**

| position   | if no structural or numerical zero, position is -1, otherwise if A(j,j) is missing or L(j,j) is not positive definite, position=j. |

See cusparseStatus_t for the description of the return status

### 11.2. Incomplete LU Factorization: level 0

Different algorithms for ilu0 are discussed in this section.

#### 11.2.1. cusparse<t>csrilu02_numericBoost()

```c
cusparseStatus_t
cusparseScsrilu02_numericBoost(cusparseHandle_t handle,
                       csrilu02Info_t info,
                       int enable_boost,
                       double* tol,
                       float* boost_val)

cusparseStatus_t
cusparseDcsrilu02_numericBoost(cusparseHandle_t handle,
                       csrilu02Info_t info,
                       int enable_boost,
                       double* tol,
                       double* boost_val)

cusparseStatus_t
cusparseCcsrilu02_numericBoost(cusparseHandle_t handle,
                       csrilu02Info_t info,
                       int enable_boost,
                       double* tol,
                       cuComplex* boost_val)

cusparseStatus_t
cusparseZcsrilu02_numericBoost(cusparseHandle_t handle,
                       csrilu02Info_t info,
                       int enable_boost,
```
The user can use a boost value to replace a numerical value in incomplete LU factorization. The tol is used to determine a numerical zero, and the boost_val is used to replace a numerical zero. The behavior is

\[
\text{if } \text{tol} \geq \text{fabs}(A(j,j)) \text{, then } A(j,j) = \text{boost_val}.
\]

To enable a boost value, the user has to set parameter enable_boost to 1 before calling `csrilu02()`. To disable a boost value, the user can call `csrilu02_numericBoost()` again with parameter `enable_boost=0`.

If `enable_boost=0`, tol and boost_val are ignored.

Both tol and boost_val can be in the host memory or device memory. The user can set the proper mode with `cusparseSetPointerMode()`.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using <code>cusparseCreateCsrilu02Info()</code>.</td>
</tr>
<tr>
<td>enable_boost</td>
<td>disable boost by <code>enable_boost=0</code>; otherwise, boost is enabled.</td>
</tr>
<tr>
<td>tol</td>
<td>tolerance to determine a numerical zero.</td>
</tr>
<tr>
<td>boost_val</td>
<td>boost value to replace a numerical zero.</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

### 11.2.2. cusparse<t>csrilu02_bufferSize()
This function returns size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting:

\[ A \approx LU \]

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

The buffer size depends on the dimension \( m \) and \( \text{nnz} \), the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call \( \text{csrilu02_bufferSize()} \) again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( m )</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{nnz} )</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>( \text{csrValA} )</td>
<td>(&lt;\text{type}&gt;\text{array of }\text{nnz} (= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)))) nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>
### 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 $\text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)$ column indices of the nonzero elements of matrix $A$.

### Output

#### info
record internal states based on different algorithms.

#### pBufferSizeInBytes
number of bytes of the buffer used in $\text{csrilu02\_analysis()}$ and $\text{csrilu02()}$.

See [cusparseStatus_t](#) for the description of the return status.

## 11.2.3. $\text{cusparse<t>csrilu02\_analysis()}$

```c
cusparseStatus_t cusparseScsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const float* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer);
```

```c
cusparseStatus_t cusparseDcsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const double* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer);
```

```c
cusparseStatus_t cusparseCcsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const cuComplex* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer);
```

```c
cusparseStatus_t cusparseZcsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const cuDoubleComplex* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer);
```
This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

\[ A \approx LU \]

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

This function requires the buffer size returned by \text{csrilu02_bufferSize()}\). The address of \( \text{pBuffer} \) must be a multiple of 128 bytes. If not, \text{CUSPARSE_STATUS_INVALID_VALUE} is returned.

Function \text{csrilu02_analysis()} reports a structural zero and computes level information stored in the opaque structure \( \text{info} \). The level information can extract more parallelism during incomplete LU factorization; however \text{csrilu02()} can be done without level information. To disable level information, the user must specify the policy of \text{csrilu02()} as \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL}.

It is the user’s choice whether to call \text{csrilu02()} if \text{csrilu02_analysis()} reports a structural zero. In this case, the user can still call \text{csrilu02()}\), which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzeros of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are \text{CUSPARSE_INDEX_BASE_ZERO} and \text{CUSPARSE_INDEX_BASE_ONE}.</td>
</tr>
<tr>
<td>csrValA</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz} = \text{csrRowPtrA}(\text{m}) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of ( \text{m} + 1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of ( \text{nnz} = \text{csrRowPtrA}(\text{m}) - \text{csrRowPtrA}(0) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>
info structure initialized using cusparseCreateCsrilu02Info().

Supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.

pBuffer buffer allocated by the user, the size is returned by csrilu02_bufferSize().

Output

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

See cusparseStatus_t for the description of the return status

11.2.4. cusparse<t>csrilu02()
This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

\[ A \approx LU \]

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

This function requires a buffer size returned by \text{csrilu02_bufferSize()}\. The address of \( \text{pBuffer} \) must be a multiple of 128 bytes. If not, \text{CUSPARSE_STATUS_INVALID_VALUE} is returned.

The matrix type must be \text{CUSPARSE_MATRIX_TYPE_GENERAL}. The fill mode and diagonal type are ignored.

Although \text{csrilu02()} can be done without level information, the user still needs to be aware of consistency. If \text{csrilu02_analysis()} is called with policy \text{CUSPARSE_SOLVE_POLICY_USE_LEVEL}, \text{csrilu02()} can be run with or without levels. On the other hand, if \text{csrilu02_analysis()} is called with \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL}, \text{csrilu02()} can only accept \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL}; otherwise, \text{CUSPARSE_STATUS_INVALID_VALUE} is returned.

Function \text{csrilu02()} reports the first numerical zero, including a structural zero. The user must call \text{cusparseXcsrilu02_zeroPivot()} to know where the numerical zero is.

For example, suppose \( A \) is a real \( m \times m \) matrix, the following code solves precondition system \( M y = x \) where \( M \) is the product of LU factors \( L \) and \( U \).

```c
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
csrilu02Info_t info_M = 0;
csrsv2Info_t info_L = 0;
csrsv2Info_t info_U = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
```
const cusparseOperation_t trans_L = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U = CUSPARSE_OPERATION_NON_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);
cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for csrilu02 and two info's for csrsv2
 cusparseCreateCsrilu02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_U);

// step 3: query how much memory used in csrilu02 and csrsv2, and allocate the buffer

cusparseDcsrilu02_bufferSize(handle, m, nnz,
   descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &pBufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
   descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_U, m, nnz,
   descr_U, d_csrVal, d_csrRowPtr, d_csrColInd, info_U, &pBufferSize_U);

pBufferSize = max(pBufferSize_M, max(pBufferSize_L, pBufferSize_U));

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

// step 4: perform analysis of incomplete Cholesky on M
//        perform analysis of triangular solve on L
//        perform analysis of triangular solve on U
// The lower(upper) triangular part of M has the same sparsity pattern as L(U),
// we can do analysis of csrilu0 and csrsv2 simultaneously.

cusparseDcsrilu02_analysis(handle, m, nnz, descr_M,
   d_csrVal, d_csrRowPtr, d_csrColInd, info_M, policy_M, pBuffer);
status = cusparseXcsrilu02_zeroPivot(handle, info_M, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status) {
   printf("A(%d,%d) is missing\n", structural_zero, structural_zero);
}
cusparseDcsrsv2_analysis(handle, trans_L, m, nnz, descr_L,
   d_csrVal, d_csrRowPtr, d_csrColInd, info_L, policy_L, pBuffer);
cusparseDcsrsv2_analysis(handle, trans_U, m, nnz, descr_U,
   d_csrVal, d_csrRowPtr, d_csrColInd, info_U, policy_U, pBuffer);
The function supports the following properties if pBuffer != NULL

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows and columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzeros of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA_valM</td>
<td>&lt;type&gt; array of nnz (= csrRowPtrA(m) - csrRowPtrA(0)) nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of m + 1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of nnz (= csrRowPtrA(m) - csrRowPtrA(0)) column indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td>info</td>
<td>structure with information collected during the analysis phase [that should have been passed to the solve phase unchanged].</td>
</tr>
</tbody>
</table>
### 11.2.5. `cusparseXcsrilu02_zeroPivot()`

```c
 cusparseStatus_t
cusparseXcsrilu02_zeroPivot(cusparseHandle_t handle,
                           csrilu02Info_t info,
                           int* position)
```

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

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

Function `cusparseXcsrilu02_zeroPivot()` is a blocking call. It calls `cudaDeviceSynchronize()` to make sure all previous kernels are done.

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

- The routine requires no extra storage
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

### Input

| handle | handle to the cuSPARSE library context. |
| info | info contains structural zero or numerical zero if the user already called `csrilu02_analysis()` or `csrilu02()`. |

### Output

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

See `cusparseStatus_t` for the description of the return status

### 11.2.6. `cusparse<t>bsrilu02_numericBoost()`

```c
 cusparseStatus_t
```

See `cusparseStatus_t` for the description of the return status
cusparseSbsrilu02_numericBoost(cusparseHandle_t handle, 
bsrilu02Info_t info, 
int enable_boost, 
double* tol, 
float* boost_val)
cusparseStatus_t

cusparseDbsrilu02_numericBoost(cusparseHandle_t handle, 
bsrilu02Info_t info, 
int enable_boost, 
double* tol, 
double* boost_val)
cusparseStatus_t

cusparseCbsrilu02_numericBoost(cusparseHandle_t handle, 
bsrilu02Info_t info, 
int enable_boost, 
double* tol, 
cuComplex* boost_val)
cusparseStatus_t

cusparseZbsrilu02_numericBoost(cusparseHandle_t handle, 
bsrilu02Info_t info, 
int enable_boost, 
double* tol, 
cuDoubleComplex* boost_val)
cusparseStatus_t

The user can use a boost value to replace a numerical value in incomplete LU factorization. Parameter tol is used to determine a numerical zero, and boost_val is used to replace a numerical zero. The behavior is as follows:

if tol >= fabs(A(j,j)), then reset each diagonal element of block A(j,j) by boost_val.

To enable a boost value, the user sets parameter enable_boost to 1 before calling bsrilu02(). To disable the boost value, the user can call bsrilu02_numericBoost() with parameter enable_boost=0.

If enable_boost=0, tol and boost_val are ignored.

Both tol and boost_val can be in host memory or device memory. The user can set the proper mode with cusparseSetPointerMode().

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using cusparseCreateBsrilu02Info().</td>
</tr>
<tr>
<td>enable_boost</td>
<td>disable boost by setting enable_boost=0. Otherwise, boost is enabled.</td>
</tr>
<tr>
<td>tol</td>
<td>tolerance to determine a numerical zero.</td>
</tr>
<tr>
<td>boost_val</td>
<td>boost value to replace a numerical zero.</td>
</tr>
</tbody>
</table>
See `cusparseStatus_t` for the description of the return status

### 11.2.7. `cusparse<t>bsrilu02_bufferSize()`

```c
 cusparseStatus_t
cusparseSbsrilu02_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dirA,
    int mb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    float *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    bsrilu02Info_t info,
    int *pBufferSizeInBytes);
```

```c
 cusparseStatus_t
cusparseDbsrilu02_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dirA,
    int mb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    double *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    bsrilu02Info_t info,
    int *pBufferSizeInBytes);
```

```c
 cusparseStatus_t
cusparseCbsrilu02_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dirA,
    int mb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    cuComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    bsrilu02Info_t info,
    int *pBufferSizeInBytes);
```

```c
 cusparseStatus_t
cusparseZbsrilu02_bufferSize(cusparseHandle_t handle,
    cusparseDirection_t dirA,
    int mb,
    int nnzb,
    const cusparseMatDescr_t descrA,
    cuDoubleComplex *bsrValA,
    const int *bsrRowPtrA,
    const int *bsrColIndA,
    int blockDim,
    bsrilu02Info_t info,
    int *pBufferSizeInBytes);
```

This function returns the size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting.
$A \approx LU$

$A$ is an $(mb \times blockDim) \times (mb \times blockDim)$ sparse matrix that is defined in BSR storage format by the three arrays $bsrValA, bsrRowPtrA, \text{ and } bsrColIndA$.

The buffer size depends on the dimensions of $mb, blockDim$, and the number of nonzero blocks of the matrix $nnzb$. If the user changes the matrix, it is necessary to call $bsrilu02\_bufferSize()$ again to have the correct buffer size; otherwise, a segmentation fault may occur.

**Input**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks, either $\text{CUSPARSE_DIRECTION_ROW}$ or $\text{CUSPARSE_DIRECTION_COLUMN}$</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows and columns of matrix $A$.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix $A$.</td>
</tr>
</tbody>
</table>
| descrA      | the descriptor of matrix $A$. The supported matrix type is $\text{CUSPARSE\_MATRIX\_TYPE\_GENERAL}$.

Also, the supported index bases are $\text{CUSPARSE\_INDEX\_BASE\_ZERO}$ and $\text{CUSPARSE\_INDEX\_BASE\_ONE}$.

| bsrValA  | $<\text{type}>$ array of $nnzb ( = bsrRowPtrA(mb) - bsrRowPtrA(0)) \text{ nonzero blocks of matrix }A \text{.}$ |
| bsrRowPtrA | integer array of $mb + 1$ elements that contains the start of every block row and the end of the last block row plus one. |
| bsrColIndA | integer array of $nnzb ( = bsrRowPtrA(mb) - bsrRowPtrA(0)) \text{ column indices of the nonzero blocks of matrix }A \text{.}$ |
| blockDim | block dimension of sparse matrix $A$, larger than zero.                     |

**Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>record internal states based on different algorithms.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>number of bytes of the buffer used in $bsrilu02_analysis()$ and $bsrilu02()$.</td>
</tr>
</tbody>
</table>

**Status Returned**

- $\text{CUSPARSE\_STATUS\_SUCCESS}$: the operation completed successfully.
- $\text{CUSPARSE\_STATUS\_NOT\_INITIALIZED}$: the library was not initialized.
- $\text{CUSPARSE\_STATUS\_ALLOC\_FAILED}$: the resources could not be allocated.
- $\text{CUSPARSE\_STATUS\_INVALID\_VALUE}$: invalid parameters were passed ($mb, nnzb <= 0$), base index is not 0 or 1.
- $\text{CUSPARSE\_STATUS\_ARCH\_MISMATCH}$: the device only supports compute capability 2.0 and above.
<table>
<thead>
<tr>
<th>CUSPARSE_STATUS_INTERNAL_ERROR</th>
<th>an internal operation failed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED</td>
<td>the matrix type is not supported.</td>
</tr>
</tbody>
</table>

## 11.2.8. `cusparse<t>bsrilu02_analysis()`

```c
CUSPARSE_STATUS_t
cusparseSbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           float* bsrValA,
                           const int* bsrRowPtrA,
                           const int* bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void* pBuffer)
```

```c
CUSPARSE_STATUS_t
cusparseDbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           double* bsrValA,
                           const int* bsrRowPtrA,
                           const int* bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void* pBuffer)
```

```c
CUSPARSE_STATUS_t
cusparseCbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex* bsrValA,
                           const int* bsrRowPtrA,
                           const int* bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void* pBuffer)
```

```c
CUSPARSE_STATUS_t
cusparseZbsrilu02_analysis(cusparseHandle_t handle,
                           cusparseDirection_t dirA,
                           int mb,
                           int nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex* bsrValA,
                           const int* bsrRowPtrA,
                           const int* bsrColIndA,
                           int blockDim,
                           bsrilu02Info_t info,
                           cusparseSolvePolicy_t policy,
                           void* pBuffer)
```
This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

\[ A \approx LU \]

\( A \) is an \((mb*\text{blockDim}) \times (mb*\text{blockDim})\) sparse matrix that is defined in BSR storage format by the three arrays \( bsrValA \), \( bsrRowPtrA \), and \( bsrColIndA \). The block in BSR format is of size \( \text{blockDim} \times \text{blockDim} \), stored as column-major or row-major as determined by parameter \( \text{dirA} \), which is either \( \text{CUSPARSE_DIRECION_COLUMN} \) or \( \text{CUSPARSE_DIRECION_ROW} \). The matrix type must be \( \text{CUSPARSE_MATRIX_TYPE_GENERAL} \), and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by \( \text{bsrilu02BufferSize()} \). The address of \( \text{pBuffer} \) must be multiple of 128 bytes. If it is not, \( \text{CUSPARSE_STATUS_INVALID_VALUE} \) is returned.

Function \( \text{bsrilu02_analysis()} \) reports a structural zero and computes level information stored in the opaque structure \( \text{info} \). The level information can extract more parallelism during incomplete LU factorization. However \( \text{bsrilu02()} \) can be done without level information. To disable level information, the user needs to specify the parameter \( \text{policy} \) of \( \text{bsrilu02[\_analysis\|]} \) as \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \).

Function \( \text{bsrilu02_analysis()} \) always reports the first structural zero, even with parameter \( \text{policy} \) as \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \). The user must call \( \text{cusparseXbsrilu02_zeroPivot()} \) to know where the structural zero is.

It is the user’s choice whether to call \( \text{bsrilu02()} \) if \( \text{bsrilu02_analysis()} \) reports a structural zero. In this case, the user can still call \( \text{bsrilu02()} \), which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirA</td>
<td>storage format of blocks, either CUSPARSE_DIRECION_ROW or CUSPARSE_DIRECION_COLUMN.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows and block columns of matrix ( A ).</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>(&lt;\text{type}&gt; ) array of ( \text{nnzb} = bsrRowPtrA(mb) - bsrRowPtrA(0) ) nonzero blocks of matrix ( A ).</td>
</tr>
</tbody>
</table>
### cusparsePreconditioners Reference

**bsrRowPtrA**  
integer array of \( mb + 1 \) elements that contains the start of every block row and the end of the last block row plus one.

**bsrColIndA**  
integer array of \( nnzb = bsrRowPtrA(mb) - bsrRowPtrA(0) \) column indices of the nonzero blocks of matrix A.

**blockDim**  
block dimension of sparse matrix A, larger than zero.

**info**  
structure initialized using cusparseCreateBsrilu02Info().

**policy**  
the supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.

**pBuffer**  
buffer allocated by the user, the size is returned by bsrilu02_bufferSize().

### Output

**info**  
structure filled with information collected during the analysis phase [that should be passed to the solve phase unchanged].

See **cusparseStatus_t** for the description of the return status.

### 11.2.9. cusparse<t>bsrilu02()**

```c
cusparseStatus_t cusparseSbsrilu02(cusparseHandle_t         handle,
                                  cusparseDirection_t      dirA,
                                  int                      mb,
                                  int                      nnzb,
                                  const cusparseMatDescr_t descry,
                                  float*                   bsrValA,
                                  const int*               bsrRowPtrA,
                                  const int*               bsrColIndA,
                                  int                      blockDim,
                                  bsrilu02Info_t           info,
                                  cusparseSolvePolicy_t    policy,
                                  void*                    pBuffer)
```

```c
cusparseStatus_t cusparseDbisluf02(cusparseHandle_t         handle,
                                  cusparseDirection_t      dirA,
                                  int                      mb,
                                  int                      nnzb,
                                  const cusparseMatDescr_t descry,
                                  double*                  bsrValA,
                                  const int*               bsrRowPtrA,
                                  const int*               bsrColIndA,
                                  int                      blockDim,
                                  bsrilu02Info_t           info,
                                  cusparseSolvePolicy_t    policy,
                                  void*                    pBuffer)
```
This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting

\[ A \approx LU \]

\( A \) is an \((mb*blockDim)\times(mb*blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays \( bsrValA, bsrRowPtrA, \) and \( bsrColIndA \). The block in BSR format is of size \( blockDim*blockDim \), stored as column-major or row-major determined by parameter \( dirA \), which is either \text{CUSPARSE\_DIRECTION\_COLUMN} or \text{CUSPARSE\_DIRECTION\_ROW}. The matrix type must be \text{CUSPARSE\_MATRIX\_TYPE\_GENERAL}, and the fill mode and diagonal type are ignored.

Function \text{bsrilu02}() supports an arbitrary \( blockDim \).

This function requires a buffer size returned by \text{bsrilu02\_bufferSize}(). The address of \( pBuffer \) must be a multiple of 128 bytes. If it is not, \text{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

Although \text{bsrilu02}() can be used without level information, the user must be aware of consistency. If \text{bsrilu02\_analysis}() is called with policy \text{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL}, \text{bsrilu02}() can be run with or without levels. On the other hand, if \text{bsrilu02\_analysis}() is called with \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}, \text{bsrilu02}() can only accept \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}; otherwise, \text{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

Function \text{bsrilu02}() has the same behavior as \text{csrilu02}(). That is, \( bsr2csr(bsrilu02(A)) = csrilu02(bsr2csr(A)) \). The numerical zero of \text{csrilu02}() means there exists some zero \( U(j,j) \). The numerical zero of \text{bsrilu02}() means there exists some block \( U(j,j) \) that is not invertible.

Function \text{bsrilu02} reports the first numerical zero, including a structural zero. The user must call \text{cusparseXbsrilu02\_zeroPivot}() to know where the numerical zero is.
For example, suppose \( A \) is a real \( m \times m \) matrix where \( m = m_b \times \text{blockDim} \). The following code solves precondition system \( M \cdot y = x \), where \( M \) is the product of LU factors \( L \) and \( U \).

```c
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is m_b, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
bsrilu02Info_t info_M = 0;
bsrsv2Info_t info_L = 0;
bsrsv2Info_t info_U = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);

cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsrilu02 and two info’s for bsrsv2
cusparseCreateBsrilu02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_U);

// step 3: query how much memory used in bsrilu02 and bsrsv2, and allocate the
// buffer
cusparseDbsrilu02_bufferSize(handle, dir, mb, nnzb,
   descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M, &pBufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
   descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L, &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_U, mb, nnzb,
   descr_U, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_U, &pBufferSize_U);

pBufferSize = max(pBufferSize_M, max(pBufferSize_L, pBufferSize_U));

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

// step 4: perform analysis of incomplete LU factorization on M
   perform analysis of triangular solve on L
   perform analysis of triangular solve on U
// The lower(upper) triangular part of M has the same sparsity pattern as L(U),
// we can do analysis of bsrilu0 and bsrsv2 simultaneously.
cusparseDbsrilu02_analysis(handle, dir, mb, nnzb, descr_M,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M,
   policy_M, pBuffer);
status = cusparseXbsrilu02_zeroPivot(handle, info_M, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status)
    printf("A(%d,%d) is missing\n", structural_zero, structural_zero);

cusparseDbsrsv2_analysis(handle, dir, trans_L, mb, nnzb, descr_L,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim,
   info_L, policy_L, pBuffer);
cusparseDbsrsv2_analysis(handle, dir, trans_U, mb, nnzb, descr_U,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim,
   info_U, policy_U, pBuffer);

// step 5: M = L * U
cusparseDbsrilu02(handle, dir, mb, nnzb, descr_M,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M, policy_M, pBuffer);
status = cusparseXbsrilu02_zeroPivot(handle, info_M, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status)
    printf("block U(%d,%d) is not invertible\n", numerical_zero, numerical_zero);

// step 6: solve L*z = x
cusparseDbsrsv2_solve(handle, dir, trans_L, mb, nnzb, &alpha, descr_L,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
   d_x, d_z, policy_L, pBuffer);

// step 7: solve U*y = z
cusparseDbsrsv2_solve(handle, dir, trans_U, mb, nnzb, &alpha, descr_U,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_U,
   d_z, d_y, policy_U, pBuffer);

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyMatDescr(descr_M);
cusparseDestroyMatDescr(descr_L);
cusparseDestroyMatDescr(descr_U);
cusparseDestroyBsrilu02Info(info_M);
cusparseDestroyBsrsv2Info(info_L);
cusparseDestroyBsrsv2Info(info_U);
cusparseDestroy(handle);

The function supports the following properties if pBuffer != NULL

- The routine requires no extra storage
- The routine supports asynchronous execution
The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>storage format of blocks: either CUSPARSE_DIRECTION_ROW or</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows and block columns of matrix A.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) nonzero blocks of</td>
</tr>
<tr>
<td></td>
<td>matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of mb + 1 elements that contains the start of every block</td>
</tr>
<tr>
<td></td>
<td>row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0)) column indices of</td>
</tr>
<tr>
<td></td>
<td>the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A; must be larger than zero.</td>
</tr>
<tr>
<td>info</td>
<td>structure with information collected during the analysis phase [that should</td>
</tr>
<tr>
<td></td>
<td>have been passed to the solve phase unchanged].</td>
</tr>
<tr>
<td>policy</td>
<td>the supported policies are</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user; the size is returned by bsrilu02_bufferSize().</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; matrix containing the incomplete-LU lower and upper triangular factors.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 11.2.10. cusparseXbsrilu02_zeroPivot()

```
cusparseStatus_t
cusparseXbsrilu02_zeroPivot(cusparseHandle_t handle,
                           bsrilu02Info_t info,
                           int* position)
```
If the returned error code is CUSPARSE_STATUS_ZERO_PIVOT, position=j means A(j,j) has either a structural zero or a numerical zero (the block is not invertible). Otherwise position=-1.

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

Function cusparseXbsrilu02_zeroPivot() is a blocking call. It calls cudaDeviceSynchronize() to make sure all previous kernels are done.

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

- The routine requires no extra storage
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>info contains structural zero or numerical zero if the user already called bsrilu02_analysis() or bsrilu02().</td>
</tr>
</tbody>
</table>

**Output**

| position | if no structural or numerical zero, position is -1; otherwise if A(j,j) is missing or U(j,j) is not invertible, position=j. |

See cusparseStatus_t for the description of the return status

### 11.3. Tridiagonal Solve

Different algorithms for tridiagonal solve are discussed in this section.

#### 11.3.1. cusparse<t>gtsv2_buffSizeExt()

```c
 cusparseStatus_t cusparseSgtsv2_bufferSizeExt(cusparseHandle_t handle, int m, int n, const float* dl, const float* d, const float* du, const float* B, int ldb, size_t* bufferSizeInBytes)
```

```c
 cusparseStatus_t cusparseDgtsv2_bufferSizeExt(cusparseHandle_t handle, int m, int n, const double* dl, ...
```
This function returns the size of the buffer used in \( gtsv2 \) which computes the solution of a tridiagonal linear system with multiple right-hand sides.

\[
A \cdot X = B
\]

The coefficient matrix \( A \) of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower \( \{d_l\} \), main \( \{d\} \), and upper \( \{d_u\} \) matrix diagonals; the right-hand sides are stored in the dense matrix \( B \). Notice that solution \( X \) overwrites right-hand-side matrix \( B \) on exit.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>the size of the linear system [must be ( \geq 3 )].</td>
</tr>
<tr>
<td>n</td>
<td>number of right-hand sides, columns of matrix ( B ).</td>
</tr>
<tr>
<td>dl</td>
<td>(&lt;\text{type}&gt;) dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td>d</td>
<td>(&lt;\text{type}&gt;) dense array containing the main diagonal of the tri-diagonal linear system.</td>
</tr>
<tr>
<td>du</td>
<td>(&lt;\text{type}&gt;) dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.</td>
</tr>
</tbody>
</table>
B | <type> dense right-hand-side array of dimensions (ldb, n).

ldb | leading dimension of B [that is ≥ max(1, m)].

Output

pBufferSizeInBytes | number of bytes of the buffer used in the gtsv2.

See `cusparseStatus_t` for the description of the return status.

### 11.3.2. cusparse<t>gtsv2()

This function computes the solution of a tridiagonal linear system 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 ($d_l$), main ($d$), and upper ($d_u$) matrix diagonals; the right-hand sides are stored in the dense matrix $B$. Notice that solution $X$ overwrites right-hand-side matrix $B$ on exit.

Assuming $A$ is of size $m$ and base-1, $d_l$, $d$ and $d_u$ are defined by the following formula:

$$d_l(i) := A(i, i-1) \text{ for } i=1,2,\ldots,m$$

The first element of $d_l$ is out-of-bound ($d_l(1) := A(1,0)$), so $d_l(1) = 0$.

$$d(i) = A(i,i) \text{ for } i=1,2,\ldots,m$$

$$d_u(i) = A(i,i+1) \text{ for } i=1,2,\ldots,m$$

The last element of $d_u$ is out-of-bound ($d_u(m) := A(m,m+1)$), so $d_u(m) = 0$.

The routine does perform pivoting, which usually results in more accurate and more stable results than cusparse<gtsv_nopivot>() or cusparse<gtsv2_nopivot>() at the expense of some execution time.

This function requires a buffer size returned by gtsv2_bufferSizeExt(). The address of pBuffer must be multiple of 128 bytes. If it is not, CUSPARSE_STATUS_INVALID_VALUE is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>the size of the linear system (must be $\geq 3$).</td>
</tr>
<tr>
<td>n</td>
<td>number of right-hand sides, columns of matrix $B$.</td>
</tr>
<tr>
<td>$d_l$</td>
<td>$&lt;\text{type}&gt;$ dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td>$d$</td>
<td>$&lt;\text{type}&gt;$ dense array containing the main diagonal of the tri-diagonal linear system.</td>
</tr>
<tr>
<td>$d_u$</td>
<td>$&lt;\text{type}&gt;$ dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.</td>
</tr>
<tr>
<td>$B$</td>
<td>$&lt;\text{type}&gt;$ dense right-hand-side array of dimensions $(\text{ldb}, n)$.</td>
</tr>
<tr>
<td>$\text{ldb}$</td>
<td>leading dimension of $B$ (that is $\geq \max(1, m)$).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is return by gtsv2_bufferSizeExt.</td>
</tr>
</tbody>
</table>

### Output
See \texttt{cusparseStatus_t} for the description of the return status

\section*{11.3.3. \texttt{cusparse\langle t\rangle gtsv2\_nopivot\_bufferSizeExt()}}

\begin{verbatim}
cusparseStatus_t
cusparseSgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    const float* dl,
    const float* d,
    const float* du,
    const float* B,
    int ldb,
    size_t* bufferSizeInBytes)
cusparseStatus_t
 cusparseDgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    const double* dl,
    const double* d,
    const double* du,
    const double* B,
    int ldb,
    size_t* bufferSizeInBytes)
cusparseStatus_t
 cusparseCgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    const cuComplex* dl,
    const cuComplex* d,
    const cuComplex* du,
    const cuComplex* B,
    int ldb,
    size_t* bufferSizeInBytes)
cusparseStatus_t
 cusparseZgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex* dl,
    const cuDoubleComplex* d,
    const cuDoubleComplex* du,
    const cuDoubleComplex* B,
    int ldb,
    size_t* bufferSizeInBytes)
\end{verbatim}

This function returns the size of the buffer used in \texttt{gtsv2\_nopivot} which computes the solution of a tridiagonal linear system with multiple right-hand sides.

\[ A \times X = B \]
The coefficient matrix $A$ of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower ($dl$), main ($d$), and upper ($du$) matrix diagonals; the right-hand sides are stored in the dense matrix $B$. Notice that solution $X$ overwrites right-hand-side matrix $B$ on exit.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>the size of the linear system (must be $\geq 3$).</td>
</tr>
<tr>
<td>n</td>
<td>number of right-hand sides, columns of matrix $B$.</td>
</tr>
<tr>
<td>dl</td>
<td>$&lt;$type$&gt;$ dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td>d</td>
<td>$&lt;$type$&gt;$ dense array containing the main diagonal of the tri-diagonal linear system.</td>
</tr>
<tr>
<td>du</td>
<td>$&lt;$type$&gt;$ dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.</td>
</tr>
<tr>
<td>B</td>
<td>$&lt;$type$&gt;$ dense right-hand-side array of dimensions $(ldb, n)$.</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of $B$. (that is $\geq \max(1, m)$).</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pBufferSizeInBytes</td>
<td>number of bytes of the buffer used in the gtsv2_nopivot.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 11.3.4. cusparse<t>gtsv2_nopivot()

```c

cusparseStatus_t
cusparseSgtsv2_nopivot(cusparseHandle_t handle, int m, int n, const float* dl, const float* d, const float* du, float* B, int ldb, void* pBuffer)

cusparseStatus_t
cusparseDgtsv2_nopivot(cusparseHandle_t handle, int m, int n, const double* dl, const double* d, const double* du, double* B, int ldb, void* pBuffer)
```
This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

\[ A \times X = B \]

The coefficient matrix \( A \) of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower \( \{ dl \} \), main \( \{ d \} \), and upper \( \{ du \} \) matrix diagonals; the right-hand sides are stored in the dense matrix \( B \). Notice that solution \( X \) overwrites right-hand-side matrix \( B \) on exit.

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

This function requires a buffer size returned by \( \text{gtsv2\_nopivot\_bufferSizeExt()} \).

The address of \( p\text{Buffer} \) must be multiple of 128 bytes. If it is not, \( \text{CUSPARSE\_STATUS\_INVALID\_VALUE} \) is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{handle} )</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( m )</td>
<td>the size of the linear system (must be ( \geq 3 )).</td>
</tr>
<tr>
<td>( n )</td>
<td>number of right-hand sides, columns of matrix ( B ).</td>
</tr>
</tbody>
</table>
### dl
<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.

### d
<type> dense array containing the main diagonal of the tri-diagonal linear system.

### du
<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.

### B
<type> dense right-hand-side array of dimensions (ldb, n).

### ldb
leading dimension of B. (that is > max(1, m))

### pBuffer
buffer allocated by the user, the size is return by gtsv2_nopivot_bufferSizeExt.

### Output

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

See <code>cusparseStatus_t</code> for the description of the return status

## 11.4. Batched Tridiagonal Solve

Different algorithms for batched tridiagonal solve are discussed in this section.

### 11.4.1. cusparse<t>gtsv2StridedBatch_bufferSizeExt()

cusparseStatus_t
cusparseSgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
   int m,
   const float* dl,
   const float* d,
   const float* du,
   const float* x,
   int batchCount,
   int batchStride,
   size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseDgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
   int m,
   const double* dl,
   const double* d,
   const double* du,
   const double* x,
   int batchCount,
   int batchStride,
   size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
   int m,
This function returns the size of the buffer used in `gtsv2StridedBatch` which computes the solution of multiple tridiagonal linear systems for \( i=0,\ldots,\text{batchCount} \):

\[
A^{(i)} \cdot y^{(i)} = x^{(i)}
\]

The coefficient matrix \( A \) of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower \( \{d_l\} \), main \( \{d\} \), and upper \( \{d_u\} \) matrix diagonals; the right-hand sides are stored in the dense matrix \( X \). Notice that solution \( Y \) overwrites right-hand-side matrix \( X \) on exit. The different matrices are assumed to be of the same size and are stored with a fixed \( \text{batchStride} \) in memory.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>Handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The size of the linear system (must be ( \geq 3 )).</td>
</tr>
<tr>
<td><code>dl</code></td>
<td><code>&lt;type&gt;</code> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal ( d_l^{(i)} ) that corresponds to the ( i )-th linear system starts at location ( d_l + \text{batchStride} \times i ) in memory. Also, the first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td><code>d</code></td>
<td><code>&lt;type&gt;</code> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal ( d^{(i)} ) that corresponds to the ( i )-th linear system starts at location ( d + \text{batchStride} \times i ) in memory.</td>
</tr>
<tr>
<td><code>du</code></td>
<td><code>&lt;type&gt;</code> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal ( d_u^{(i)} ) that corresponds to the ( i )-th linear system starts at location ( d_u + \text{batchStride} \times i ) in memory.</td>
</tr>
</tbody>
</table>
system starts at location $du + du\cdot i$ in memory. Also, the last element of each upper diagonal must be zero.

$x$<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^i$ that corresponds to the $i^{th}$ linear system starts at location $x + x\cdot i$ in memory.

batchCount number of systems to solve.

batchStride stride (number of elements) that separates the vectors of every system (must be at least $m$).

Output

pBufferSizeInBytes number of bytes of the buffer used in the gtsv2StridedBatch.

See [cusparseStatus_t](#) for the description of the return status

### 11.4.2. cusparse<t>gtsv2StridedBatch()

```
cusparseStatus_t cusparseSgtsv2StridedBatch(cusparseHandle_t handle, 
    int m, 
    const float* dl, 
    const float* d, 
    const float* du, 
    float* x, 
    int batchCount, 
    int batchStride, 
    void* pBuffer)
```

```
cusparseStatus_t cusparseDgtsv2StridedBatch(cusparseHandle_t handle, 
    int m, 
    const double* dl, 
    const double* d, 
    const double* du, 
    double* x, 
    int batchCount, 
    int batchStride, 
    void* pBuffer)
```

```
cusparseStatus_t cusparseCgtsv2StridedBatch(cusparseHandle_t handle, 
    int m, 
    const cuComplex* dl, 
    const cuComplex* d, 
    const cuComplex* du, 
    cuComplex* x, 
    int batchCount, 
    int batchStride, 
    void* pBuffer)
```

```
cusparseStatus_t cusparseZgtsv2StridedBatch(cusparseHandle_t handle, 
```

```
```
This function computes the solution of multiple tridiagonal linear systems for $i=0, \ldots, \text{batchCount}$:

$$A^i x^i = y^i$$

The coefficient matrix $A$ of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower ($dl$), main ($d$), and upper ($du$) matrix diagonals; the right-hand sides are stored in the dense matrix $X$. Notice that solution $Y$ overwrites right-hand-side matrix $X$ on exit. The different matrices are assumed to be of the same size and are stored with a fixed batchStride in memory.

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

This function requires a buffer size returned by \texttt{gtsv2StridedBatch\_bufferSizeExt()}. The address of \texttt{pBuffer} must be multiple of 128 bytes. If it is not, \texttt{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>the size of the linear system (must be $\geq 3$).</td>
</tr>
<tr>
<td><code>dl</code></td>
<td>&lt;type&gt; dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^i$ that corresponds to the $i^{th}$ linear system starts at location $dl + \text{batchStride} \times i$ in memory. Also, the first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td><code>d</code></td>
<td>&lt;type&gt; dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^i$ that corresponds to the $i^{th}$ linear system starts at location $d + \text{batchStride} \times i$ in memory.</td>
</tr>
<tr>
<td><code>du</code></td>
<td>&lt;type&gt; dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^i$ that corresponds to the $i^{th}$ linear system starts at location $du + \text{batchStride} \times i$ in memory.</td>
</tr>
<tr>
<td><code>x</code>, <code>y</code></td>
<td>the solution vectors $X$ and $Y$.</td>
</tr>
<tr>
<td><code>batchCount</code></td>
<td>the number of linear systems.</td>
</tr>
<tr>
<td><code>batchStride</code></td>
<td>the stride between batches.</td>
</tr>
<tr>
<td><code>pBuffer</code></td>
<td>the buffer to be used.</td>
</tr>
</tbody>
</table>
memory. Also, the last element of each upper diagonal must be zero.

\( x \)  
\(<\text{type}>\) dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side \( x^i \) that corresponds to the \( i^{th} \) linear system starts at location \( x + \text{batchStride} \times i \) in memory.

\( \text{batchCount} \)  
number of systems to solve.

\( \text{batchStride} \)  
stride (number of elements) that separates the vectors of every system [must be at least \( n \)].

\( \text{pBuffer} \)  
buffer allocated by the user, the size is return by \text{gtsv2StridedBatch_bufferSizeExt}.

\[ \text{Output} \]

\( x \)  
\(<\text{type}>\) dense array that contains the solution of the tri-diagonal linear system. The solution \( x^i \) that corresponds to the \( i^{th} \) linear system starts at location \( x + \text{batchStride} \times i \) in memory.

See \text{cusparseStatus_t} for the description of the return status

11.4.3. \text{cusparse<}t\text{>gtsvInterleavedBatch()}
```
cusparseStatus_t
cusparseZgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle, int algo, int m, const cuDoubleComplex* dl, const cuDoubleComplex* d, const cuDoubleComplex* du, const cuDoubleComplex* x, int batchCount, size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseZgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle, int algo, int m, const cuDoubleComplex* dl, const cuDoubleComplex* d, const cuDoubleComplex* du, const cuDoubleComplex* x, int batchCount, size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseSgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, float* dl, float* d, float* du, float* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseSgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, float* dl, float* d, float* du, float* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseDgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, double* dl, double* d, double* du, double* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseDgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, double* dl, double* d, double* du, double* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseCgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, cuComplex* dl, cuComplex* d, cuComplex* du, cuComplex* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseCgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, cuComplex* dl, cuComplex* d, cuComplex* du, cuComplex* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseZgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, cuDoubleComplex* dl, cuDoubleComplex* d, cuDoubleComplex* du, cuDoubleComplex* x, int batchCount, void* pBuffer)

cusparseStatus_t
cusparseZgtsvInterleavedBatch(cusparseHandle_t handle, int algo, int m, cuDoubleComplex* dl, cuDoubleComplex* d, cuDoubleComplex* du, cuDoubleComplex* x, int batchCount, void* pBuffer)
```

```
const cuComplex* x,
int              batchCount,
size_t*          pBufferSizeInBytes)
```
This function computes the solution of multiple tridiagonal linear systems for $i=0, …, \text{batchCount}$:

$$A^{(i)} \times x^{(i)} = b^{(i)}$$

The coefficient matrix $A$ of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower ($dl$), main ($d$), and upper ($du$) matrix diagonals; the right-hand sides are stored in the dense matrix $B$. Notice that solution $X$ overwrites right-hand-side matrix $B$ on exit.

Assuming $A$ is of size $m$ and base-1, $dl$, $d$ and $du$ are defined by the following formula:

\[
dl(i) := A(i, i-1) \text{ for } i=1,2,\ldots,m
\]

The first element of $dl$ is out-of-bound ($dl(1) := A(1,0)$), so $dl(1) = 0$.

\[
d(i) = A(i,i) \text{ for } i=1,2,\ldots,m
\]

\[
du(i) = A(i,i+1) \text{ for } i=1,2,\ldots,m
\]

The last element of $du$ is out-of-bound ($du(m) := A(m,m+1)$), so $du(m) = 0$.

The data layout is different from $gtsvStridedBatch$ which aggregates all matrices one after another. Instead, $gtsvInterleavedBatch$ gathers different matrices of the same element in a continuous manner. If $dl$ is regarded as a 2-D array of size $m$-by-$\text{batchCount}$, $dl(:,j)$ to store $j$-th matrix. $gtsvStridedBatch$ uses column-major while $gtsvInterleavedBatch$ uses row-major.

The routine provides three different algorithms, selected by parameter $algo$. The first algorithm is cuThomas provided by Barcelona Supercomputing Center. The second algorithm is LU with partial pivoting and last algorithm is QR. From stability perspective, cuThomas is not numerically stable because it does not have pivoting. LU with partial pivoting and QR are stable. From performance perspective, LU with partial pivoting and QR is about 10% to 20% slower than cuThomas.

This function requires a buffer size returned by $gtsvInterleavedBatch\_bufferSizeExt()$. The address of $pBuffer$ must be multiple of 128 bytes. If it is not, CUSPARSE\_STATUS\_INVALID\_VALUE is returned.

If the user prepares aggregate format, one can use $cublasXgeam$ to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>algo</td>
<td>$algo = 0$: cuThomas (unstable algorithm); $algo = 1$: LU with pivoting (stable algorithm); $algo = 2$: QR (stable algorithm)</td>
</tr>
<tr>
<td>m</td>
<td>the size of the linear system.</td>
</tr>
<tr>
<td>dl</td>
<td>&lt;type&gt; dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>d</td>
<td>&lt;type&gt; dense array containing the main diagonal of the tri-diagonal linear system.</td>
</tr>
<tr>
<td>du</td>
<td>&lt;type&gt; dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.</td>
</tr>
<tr>
<td>x</td>
<td>&lt;type&gt; dense right-hand-side array of dimensions (batchCount, n).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is return by gtsvInterleavedBatch_bufferSizeExt.</td>
</tr>
</tbody>
</table>

Output

| x                     | <type> dense solution array of dimensions (batchCount, n).                                                                         |

See **cusparsestate_t** for the description of the return status

### 11.5. Batched Pentadiagonal Solve

Different algorithms for batched pentadiagonal solve are discussed in this section.

#### 11.5.1. cusparse<t>gpsvInterleavedBatch()

```c

cusparseStatus_t
cusparseSgpvsInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                        int              algo,
                                        int              m,
                                        const float*     ds,
                                        const float*     dl,
                                        const float*     d,
                                        const float*     du,
                                        const float*     dw,
                                        const float*     x,
                                        int              batchCount,
                                        size_t*          pBufferSizeInBytes)

cusparseStatus_t
cusparseDgpvsInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                        int              algo,
                                        int              m,
                                        const double*    ds,
                                        const double*    dl,
                                        const double*    d,
                                        const double*    du,
                                        const double*    dw,
                                        const double*    x,
                                        int              batchCount,
                                        size_t*          pBufferSizeInBytes)
```
cusparseStatus_t
cusparseCgpvsInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
int algo,
int m,
const cuComplex* ds,
const cuComplex* dl,
const cuComplex* d,
const cuComplex* du,
const cuComplex* dw,
const cuComplex* x,
int batchCount,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseZgpvsInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
int algo,
int m,
const cuDoubleComplex* ds,
const cuDoubleComplex* dl,
const cuDoubleComplex* d,
const cuDoubleComplex* du,
const cuDoubleComplex* dw,
const cuDoubleComplex* x,
int batchCount,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseSgpvsInterleavedBatch(cusparseHandle_t handle,
int algo,
int m,
float* ds,
float* dl,
float* d,
float* du,
float* dw,
float* x,
int batchCount,
void* pBuffer)

cusparseStatus_t
cusparseDgpvsInterleavedBatch(cusparseHandle_t handle,
int algo,
int m,
double* ds,
double* dl,
double* d,
double* du,
double* dw,
double* x,
int batchCount,
void* pBuffer)

cusparseStatus_t
cusparseCgpvsInterleavedBatch(cusparseHandle_t handle,
int algo,
int m,
cuComplex* ds,
This function computes the solution of multiple penta-diagonal linear systems for \(i=0, \ldots, \text{batchCount}\):

\[
A^{(i)} x^{(i)} = b^{(i)}
\]

The coefficient matrix \(A\) of each of these penta-diagonal linear system is defined with five vectors corresponding to its lower \(d_s, d_l\), main \(d\), and upper \(d_u, d_w\) matrix diagonals; the right-hand sides are stored in the dense matrix \(B\). Notice that solution \(X\) overwrites right-hand-side matrix \(B\) on exit.

Assuming \(A\) is of size \(m\) and base-1, \(d_s, d_l, d, d_u\) and \(d_w\) are defined by the following formula:

- \(d_s(i) := A(i, i-2)\) for \(i=1,2,\ldots,m\)
- The first two elements of \(d_s\) is out-of-bound \((d_s(1) := A(1,-1), d_s(2) := A(2,0))\), so \(d_s(1) = 0\) and \(d_s(2) = 0\).
- \(d_l(i) := A(i, i-1)\) for \(i=1,2,\ldots,m\)
- The first element of \(d_l\) is out-of-bound \((d_l(1) := A(1,0))\), so \(d_l(1) = 0\).
- \(d(i) = A(i,i)\) for \(i=1,2,\ldots,m\)
- \(d_u(i) = A(i,i+1)\) for \(i=1,2,\ldots,m\)
- The last element of \(d_u\) is out-of-bound \((d_u(m) := A(m,m+1))\), so \(d_u(m) = 0\).
- \(d_w(i) = A(i,i+2)\) for \(i=1,2,\ldots,m\)
- The last two elements of \(d_w\) is out-of-bound \((d_w(m-1) := A(m-1,m+1), d_w(m) := A(m,m+2))\), so \(d_w(m-1) = 0\) and \(d_w(m) = 0\).

The data layout is the same as \text{gtsvStridedBatch}.

The routine is numerically stable because it uses QR to solve the linear system.

This function requires a buffer size returned by \text{gpsvInterleavedBatch_bufferSizeExt}(). The address of \text{pBuffer} must be multiple of 128 bytes. If it is not, \text{CUSPARSE_STATUS_INVALID_VALUE} is returned.
Appendix section shows an example of gpsvInterleavedBatch. If the user prepares aggregate format, one can use cublasXgeam to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

The function supports the following properties if pBuffer != NULL

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>algo</td>
<td>only support algo = 0 (QR)</td>
</tr>
<tr>
<td>m</td>
<td>the size of the linear system.</td>
</tr>
<tr>
<td>ds</td>
<td>&lt;type&gt; dense array containing the lower diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The first two elements must be zero.</td>
</tr>
<tr>
<td>dl</td>
<td>&lt;type&gt; dense array containing the lower diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The first element must be zero.</td>
</tr>
<tr>
<td>d</td>
<td>&lt;type&gt; dense array containing the main diagonal of the penta-diagonal linear system.</td>
</tr>
<tr>
<td>du</td>
<td>&lt;type&gt; dense array containing the upper diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The last element must be zero.</td>
</tr>
<tr>
<td>dw</td>
<td>&lt;type&gt; dense array containing the upper diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The last two elements must be zero.</td>
</tr>
<tr>
<td>x</td>
<td>&lt;type&gt; dense right-hand-side array of dimensions (batchCount, n).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is return by gpsvInterleavedBatch_bufferSizeExt.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&lt;type&gt; dense solution array of dimensions (batchCount, n).</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status.
This chapter describes the reordering routines used to manipulate sparse matrices.

### 12.1. cudaSparse<

cusparse<t>csrColor()

```c
cusparseStatus_t
cusparseCcsrColor(cusparseHandle_t handle, int m, int nnz,
const cusparseMatDescr_t descrA, const float* csrValA,
const int* csrRowPtrA, const int* csrColIndA,
const float* fractionToColor, int* ncolors, int* coloring,
int* reordering, cusparseColorInfo_t info)

cusparseStatus_t
cusparseDcsrColor(cusparseHandle_t handle, int m, int nnz,
const cusparseMatDescr_t descrA, const double* csrValA,
const int* csrRowPtrA, const int* csrColIndA,
const double* fractionToColor, int* ncolors, int* coloring,
int* reordering, cusparseColorInfo_t info)

cusparseStatus_t
cusparseCcsrcolor(cusparseHandle_t handle, int m, int nnz,
const cusparseMatDescr_t descrA, const cuComplex* csrValA,
const int* csrRowPtrA, const int* csrColIndA,
const cuComplex* fractionToColor,
```
This function performs the coloring of the adjacency graph associated with the matrix \( A \) stored in CSR format. The coloring is an assignment of colors (integer numbers) to nodes, such that neighboring nodes have distinct colors. An approximate coloring algorithm is used in this routine, and is stopped when a certain percentage of nodes has been colored. The rest of the nodes are assigned distinct colors (an increasing sequence of integers numbers, starting from the last integer used previously). The last two auxiliary routines can be used to extract the resulting number of colors, their assignment and the associated reordering. The reordering is such that nodes that have been assigned the same color are reordered to be next to each other.

The matrix \( A \) passed to this routine, must be stored as a general matrix and have a symmetric sparsity pattern. If the matrix is nonsymmetric the user should pass \( A + A^T \) as a parameter to this routine.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>&lt;type&gt; array of ( nnz = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of ( m+1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of $nnz = csrRowPtrA(m) - csrRowPtrA(0)$ column indices of the nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>fractionToColor</td>
<td>fraction of nodes to be colored, which should be in the interval $[0.0, 1.0]$, for example 0.8 implies that 80 percent of nodes will be colored.</td>
</tr>
<tr>
<td>info</td>
<td>structure with information to be passed to the coloring.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncolors</td>
<td>The number of distinct colors used [at most the size of the matrix, but likely much smaller].</td>
</tr>
<tr>
<td>coloring</td>
<td>The resulting coloring permutation</td>
</tr>
<tr>
<td>reordering</td>
<td>The resulting reordering permutation [untouched if NULL]</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
Chapter 13. cuSPARSE Format Conversion Reference

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

cousoort, csrsort, ccsort, and csru2csr are sorting routines without malloc inside, the following table estimates the buffer size

<table>
<thead>
<tr>
<th>routine</th>
<th>buffer size</th>
<th>maximum problem size if buffer is limited by 2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>coosort</td>
<td>&gt; 16*n bytes</td>
<td>125M</td>
</tr>
<tr>
<td>csrsort or ccsort</td>
<td>&gt; 20*n bytes</td>
<td>100M</td>
</tr>
<tr>
<td>csru2csr</td>
<td>'d' &gt; 28<em>n bytes ; 'z' &gt; 36</em>n bytes</td>
<td>71M for 'd' and 55M for 'z'</td>
</tr>
</tbody>
</table>

13.1. cusparse<t>bsr2csr()

cusparseStatus_t cusparseBsr2csr(cusparseHandle_t handle, cusparseDirection_t dir, int mb, int nb, const cusparseMatDescr_t descrA, const float* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, const cusparseMatDescr_t descrC, float* csrValC, int* csrRowPtrC, int* csrColIndC)

cusparseStatus_t cusparseDbsr2csr(cusparseHandle_t handle, cusparseDirection_t dir, int mb, int nb, const cusparseMatDescr_t descrA, const double* bsrValA,
This function converts a sparse matrix in BSR format that is defined by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA`) into a sparse matrix in CSR format that is defined by arrays `csrValC`, `csrRowPtrC`, and `csrColIndC`.

Let \(m(=mb*blockDim)\) be the number of rows of \(A\) and \(n(=nb*blockDim)\) be number of columns of \(A\), then \(A\) and \(C\) are \(m*n\) sparse matrices. The BSR format of \(A\) contains \(nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])\) nonzero blocks, whereas the sparse matrix \(A\) contains \(nnz(=nnzb*blockDim*blockDim)\) elements. The user must allocate enough space for arrays `csrRowPtrC`, `csrColIndC`, and `csrValC`. The requirements are as follows:

- `csrRowPtrC` of \(m+1\) elements
- `csrValC` of \(nnz\) elements
- `csrColIndC` of \(nnz\) elements

The general procedure is as follows:

```c
// Given BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and // blocks of BSR format are stored in column-major order.

cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb*blockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz = nnzb * blockDim * blockDim; // number of elements
```
```c

cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSbsr2csr(handle, dir, mb, nb, descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockDim, descrC, csrValC, csrRowPtrC, csrColIndC);

» The routine requires no extra storage
» The routine does not support asynchronous execution if blockDim == 1
» The routine does not support CUDA graph capture if blockDim == 1

Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dir</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of sparse matrix A.</td>
</tr>
<tr>
<td>nb</td>
<td>number of block columns of sparse matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; array of nnzb<em>blockDim</em>blockDim nonzero elements of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of mb+1 elements that contains the start of every block row and the end of the last block row plus one of matrix A.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb column indices of the nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix A.</td>
</tr>
<tr>
<td>descrC</td>
<td>the descriptor of matrix C.</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValC</td>
<td>&lt;type&gt; array of nnz(=csrRowPtrC[m]-csrRowPtrC[0]) nonzero elements of matrix C.</td>
</tr>
<tr>
<td>csrRowPtrC</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one of matrix C.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>integer array of nnz column indices of the nonzero elements of matrix C.</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

13.2. cusparse<t>gebsr2gebsc()
int mb,
int nb,
int nnzb,
const float* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseDgebsr2gebsc_bufferSize(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const double* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseCgebsr2gebsc_bufferSize(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const cuComplex* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseZgebsr2gebsc_bufferSize(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const cuDoubleComplex* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseSgebsr2gebsc(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const float* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
float* bscVal,
int* bscRowInd,
int* bscColPtr,
cusparseAction_t copyValues,
cusparseIndexBase_t baseIdx,
This function can be seen as the same as `csr2csc()` when each block of size `rowBlockDim*colBlockDim` is regarded as a scalar.

This sparsity pattern of the result matrix can also be seen as the transpose of the original sparse matrix, but the memory layout of a block does not change.

The user must call `gebsr2gebsc_bufferSize()` to determine the size of the buffer required by `gebsr2gebsc()`, allocate the buffer, and pass the buffer pointer to `gebsr2gebsc()`.
The routine requires no extra storage if pBuffer != NULL
The routine does not support asynchronous execution
The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of sparse matrix A.</td>
</tr>
<tr>
<td>nb</td>
<td>number of block columns of sparse matrix A.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>bsrVal</td>
<td>&lt;type&gt; array of nnzb<em>rowBlockDim</em>colBlockDim nonzero elements of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtr</td>
<td>integer array of mb+1 elements that contains the start of every block row and the end of the last block row plus one.</td>
</tr>
<tr>
<td>bsrColInd</td>
<td>integer array of nnzb column indices of the non-zero blocks of matrix A.</td>
</tr>
<tr>
<td>rowBlockDim</td>
<td>number of rows within a block of A.</td>
</tr>
<tr>
<td>colBlockDim</td>
<td>number of columns within a block of A.</td>
</tr>
<tr>
<td>copyValues</td>
<td>CUSPARSE_ACTION_SYMBOLIC or CUSPARSE_ACTION_NUMERIC.</td>
</tr>
<tr>
<td>baseIdx</td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>pBufferSize</td>
<td>host pointer containing number of bytes of the buffer used in gebsr2gebsc().</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user; the size is return by gebsr2gebsc_bufferSize().</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bscVal</td>
<td>&lt;type&gt; array of nnzb<em>rowBlockDim</em>colBlockDim non-zero elements of matrix A. It is only filled-in if copyValues is set to CUSPARSE_ACTION_NUMERIC.</td>
</tr>
<tr>
<td>bscRowInd</td>
<td>integer array of nnzb row indices of the non-zero blocks of matrix A.</td>
</tr>
<tr>
<td>bscColPtr</td>
<td>integer array of nb+1 elements that contains the start of every block column and the end of the last block column plus one.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 13.3. cusparse<t>gebsr2gebsr()

See [cusparseStatus_t](#) for the description of the return status
### cuSPARSE Format Conversion Reference

#### cusparseSgebsr2gebsr_bufferSize

```c
cusparseStatus_t
cusparseSgebsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int mb,
int nb,
int nnzb,
const cusparseMatDescr_t descrA,
const float* bsrValA,
const int* bsrRowPtrA,
const int* bsrColIndA,
int rowBlockDimA,
int colBlockDimA,
int rowBlockDimC,
int colBlockDimC,
int* pBufferSize)
```

#### cusparseDgebsr2gebsr_bufferSize

```c
cusparseStatus_t
cusparseDgebsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int mb,
int nb,
int nnzb,
const cusparseMatDescr_t descrA,
const double* bsrValA,
const int* bsrRowPtrA,
const int* bsrColIndA,
int rowBlockDimA,
int colBlockDimA,
int rowBlockDimC,
int colBlockDimC,
int* pBufferSize)
```

#### cusparseCgebsr2gebsr_bufferSize

```c
cusparseStatus_t
cusparseCgebsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int mb,
int nb,
int nnzb,
const cusparseMatDescr_t descrA,
const cuComplex* bsrValA,
const int* bsrRowPtrA,
const int* bsrColIndA,
int rowBlockDimA,
int colBlockDimA,
int rowBlockDimC,
int colBlockDimC,
int* pBufferSize)
```

#### cusparseZgebsr2gebsr_bufferSize

```c
cusparseStatus_t
cusparseZgebsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int mb,
int nb,
int nnzb,
const cusparseMatDescr_t descrA,
const cuDoubleComplex* bsrValA,
const int* bsrRowPtrA,
const int* bsrColIndA,
int rowBlockDimA,
int colBlockDimA,
int rowBlockDimC,
int colBlockDimC,
int* pBufferSize)
```
### cusparseXgebsr2gebsrNnz

```c
int*                pBufferSize)
cusparseStatus_t   cusparseXgebsr2gebsrNnz(cusparseHandle_t handle,
                                          cusparseDirection_t dir,
                                          int mb,
                                          int nb,
                                          int nnzb,
                                          const cusparseMatDescr_t descrA,
                                          const int* bsrRowPtrA,
                                          const int* bsrColIndA,
                                          int rowBlockDimA,
                                          int colBlockDimA,
                                          const cusparseMatDescr_t descrC,
                                          int* bsrRowPtrC,
                                          int rowBlockDimC,
                                          int colBlockDimC,
                                          int* nnzTotalDevHostPtr,
                                          void* pBuffer)
```

### cusparseSgebsr2gebsr

```c
int*                pBufferSize)
cusparseStatus_t   cusparseSgebsr2gebsr(cusparseHandle_t handle,
                                          cusparseDirection_t dir,
                                          int mb,
                                          int nb,
                                          int nnzb,
                                          const cusparseMatDescr_t descrA,
                                          const float* bsrValA,
                                          const int* bsrRowPtrA,
                                          const int* bsrColIndA,
                                          int rowBlockDimA,
                                          int colBlockDimA,
                                          const cusparseMatDescr_t descrC,
                                          float* bsrValC,
                                          int* bsrRowPtrC,
                                          int* bsrColIndC,
                                          int rowBlockDimC,
                                          int colBlockDimC,
                                          void* pBuffer)
```

### cusparseDgebsr2gebsr

```c
int*                pBufferSize)
cusparseStatus_t   cusparseDgebsr2gebsr(cusparseHandle_t handle,
                                          cusparseDirection_t dir,
                                          int mb,
                                          int nb,
                                          int nnzb,
                                          const cusparseMatDescr_t descrA,
                                          const double* bsrValA,
                                          const int* bsrRowPtrA,
                                          const int* bsrColIndA,
                                          int rowBlockDimA,
                                          int colBlockDimA,
                                          const cusparseMatDescr_t descrC,
                                          double* bsrValC,
                                          int* bsrRowPtrC,
                                          int* bsrColIndC,
                                          int rowBlockDimC,
                                          int colBlockDimC,
                                          void* pBuffer)
```
This function converts a sparse matrix in general BSR format that is defined by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA` into a sparse matrix in another general BSR format that is defined by arrays `bsrValC`, `bsrRowPtrC`, and `bsrColIndC`.

If `rowBlockDimA=1` and `colBlockDimA=1`, `cusparse[S|D|C|Z]gebsr2gebsr()` is the same as `cusparse[S|D|C|Z]csr2gebsr()`.

If `rowBlockDimC=1` and `colBlockDimC=1`, `cusparse[S|D|C|Z]gebsr2gebsr()` is the same as `cusparse[S|D|C|Z]gebsr2csr()`.

A is an m*n sparse matrix where m(=mb*rowBlockDim) is the number of rows of A, and n(=nb*colBlockDim) is the number of columns of A. The general BSR format of A contains nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0]) nonzero blocks. The matrix C is also general BSR format with a different block size, rowBlockDimC*colBlockDimC. If m is not a multiple of rowBlockDimC, or n is not a multiple of colBlockDimC, zeros are filled in. The number of block rows of C is mc(=(m+rowBlockDimC-1)/rowBlockDimC). The number of block rows of C is nc(=(n+colBlockDimC-1)/colBlockDimC). The number of nonzero blocks of C is nnzc.

The implementation adopts a two-step approach to do the conversion. First, the user allocates `bsrRowPtrC` of `mc+1` elements and uses function `cusparseXgebsr2gebsrNnz()`.
to determine the number of nonzero block columns per block row of matrix \( C \). Second, the user gathers \( nnzc \) [number of non-zero block columns of matrix \( C \)] from either
(\( nnzc = *\text{nnzTotalDevHostPtr} \)) \text{ or } (\( nnzc = \text{bsrRowPtrC}[mc] - \text{bsrRowPtrC}[0] \)) and allocates \( \text{bsrValC} \) of \( nnzc \times \text{rowBlockDimC} \times \text{colBlockDimC} \) elements and \( \text{bsrColIndC} \) of \( nnzc \) integers. Finally, the function \text{cusparse[S|D|C|Z]}\text{gebsr2gebsr()}\text{ is called to complete the conversion. The user must call gebsr2gebsr_bufferSize()}\text{ to know the size of the buffer required by gebsr2gebsr(), allocate the buffer, and pass the buffer pointer to gebsr2gebsr().}

The general procedure is as follows:

```c
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and 
// blocks of BSR format are stored in column-major order.

cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzc;
int m = mb*rowBlockDimA;
int n = nb*colBlockDimA;
int mc = (m+rowBlockDimC-1)/rowBlockDimC;
int nc = (n+colBlockDimC-1)/colBlockDimC;
int bufferSize;

void *pBuffer;
cusparseSgebsr2gebsr_bufferSize(handle, dir, mb, nb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA,
rowBlockDimA, colBlockDimA,
rowBlockDimC, colBlockDimC,
&bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int)*(mc+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzc;
cusparseXgebsr2gebsrNnz(handle, dir, mb, nb, nnzb,
descrA, bsrRowPtrA, bsrColIndA,
rowBlockDimA, colBlockDimA,
descrC, bsrRowPtrC,
rowBlockDimC, colBlockDimC,
nnzTotalDevHostPtr,
pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzc = *nnzTotalDevHostPtr;
} else{  
cudaMemcpy(&nnzc, bsrRowPtrC+mc, sizeof(int), cudaMemcpyDeviceToHost);
cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzc -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzc);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDimC*colBlockDimC)*nnzc);
cusparseSgebsr2gebsr(handle, dir, mb, nb, nnzb,
descrA, bsrValA, bsrRowPtrA, bsrColIndA,
rowBlockDimA, colBlockDimA,
descrC, bsrValC, bsrRowPtrC, bsrColIndC,
rowBlockDimC, colBlockDimC,
pBuffer);
```

- The routines require no extra storage if pBuffer \(!=\) NULL
- The routines do **not** support asynchronous execution
- The routines do **not** support CUDA graph capture

**Input**

| handle | handle to the cuSPARSE library context. |
| **dir** | storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN. |
| **mb** | number of block rows of sparse matrix $A$. |
| **nb** | number of block columns of sparse matrix $A$. |
| **nnzb** | number of nonzero blocks of matrix $A$. |
| **descrA** | the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE. |
| **bsrValA** | `<type> array of nnzb*rowBlockDimA*colBlockDimA non-zero elements of matrix $A$.` |
| **bsrRowPtrA** | integer array of $mb+1$ elements that contains the start of every block row and the end of the last block row plus one of matrix $A$. |
| **bsrColIndA** | integer array of $nnzb$ column indices of the non-zero blocks of matrix $A$. |
| **rowBlockDimA** | number of rows within a block of $A$. |
| **colBlockDimA** | number of columns within a block of $A$. |
| **descrC** | the descriptor of matrix $C$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE. |
| **rowBlockDimC** | number of rows within a block of $C$. |
| **colBlockDimC** | number of columns within a block of $C$. |
| **pBufferSize** | host pointer containing number of bytes of the buffer used in gebsr2gebsr(). |
| **pBuffer** | buffer allocated by the user; the size is return by gebsr2gebsr_bufferSize(). |

**Output**

| **bsrValC** | `<type> array of nnzc*rowBlockDimC*colBlockDimC non-zero elements of matrix $C$.` |
| **bsrRowPtrC** | integer array of $mc+1$ elements that contains the start of every block row and the end of the last block row plus one of matrix $C$. |
| **bsrColIndC** | integer array of $nnzc$ block column indices of the nonzero blocks of matrix $C$. |
| **nnzTotalDevHostPtr** | total number of nonzero blocks of $C$. *$nnzTotalDevHostPtr$ is the same as $bsrRowPtrC[mc]-bsrRowPtrC[0]$. |
13.4. **cusparse<t>gebsr2csr()**

See **cusparseStatus_t** for the description of the return status.

cusparseStatus_t
cusparseSgebsr2csr(cusparseHandle_t handle, 
cusparseDirection_t dir, 
int mb, 
int nb, 
const cusparseMatDescr_t descrA, 
const float* bsrValA, 
const int* bsrRowPtrA, 
const int* bsrColIndA, 
int rowBlockDim, 
int colBlockDim, 
const cusparseMatDescr_t descrC, 
float* csrValC, 
int* csrRowPtrC, 
int* csrColIndC)

cusparseStatus_t
cusparseDgebsr2csr(cusparseHandle_t handle, 
cusparseDirection_t dir, 
int mb, 
int nb, 
const cusparseMatDescr_t descrA, 
const double* bsrValA, 
const int* bsrRowPtrA, 
const int* bsrColIndA, 
int rowBlockDim, 
int colBlockDim, 
const cusparseMatDescr_t descrC, 
double* csrValC, 
int* csrRowPtrC, 
int* csrColIndC)

cusparseStatus_t
cusparseCgebsr2csr(cusparseHandle_t handle, 
cusparseDirection_t dir, 
int mb, 
int nb, 
const cusparseMatDescr_t descrA, 
const cuComplex* bsrValA, 
const int* bsrRowPtrA, 
const int* bsrColIndA, 
int rowBlockDim, 
int colBlockDim, 
const cusparseMatDescr_t descrC, 
cuComplex* csrValC, 
int* csrRowPtrC, 
int* csrColIndC)

cusparseStatus_t
cusparseZgebsr2csr(cusparseHandle_t handle, 
cusparseDirection_t dir, 
int mb, 
int nb, 
const cusparseMatDescr_t descrA,
This function converts a sparse matrix in general BSR format that is defined by the three arrays `bsrValA`, `bsrRowPtrA`, and `bsrColIndA` into a sparse matrix in CSR format that is defined by arrays `csrValC`, `csrRowPtrC`, and `csrColIndC`.

Let $m(=mb \times \text{rowBlockDim})$ be number of rows of $A$ and $n(=nb \times \text{colBlockDim})$ be number of columns of $A$, then $A$ and $C$ are $m \times n$ sparse matrices. The general BSR format of $A$ contains $\text{nnzb}(=\text{bsrRowPtrA}[mb] - \text{bsrRowPtrA}[0])$ non-zero blocks, whereas sparse matrix $A$ contains $\text{nnz}(=\text{nnzb} \times \text{rowBlockDim} \times \text{colBlockDim})$ elements. The user must allocate enough space for arrays `csrRowPtrC`, `csrColIndC`, and `csrValC`. The requirements are as follows:

- `csrRowPtrC` of $m+1$ elements
- `csrValC` of $\text{nnz}$ elements
- `csrColIndC` of $\text{nnz}$ elements

The general procedure is as follows:

```c
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb * rowBlockDim;
int n = nb * colBlockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz = nnzb * rowBlockDim * colBlockDim; // number of elements
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSgebsr2csr(handle, dir, mb, nb,
    descrA,
    bsrValA, bsrRowPtrA, bsrColIndA,
    rowBlockDim, colBlockDim,
    descrC,
    csrValC, csrRowPtrC, csrColIndC);
```

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
</tbody>
</table>
| `dir`         | storage format of blocks, either CUSPARSE_DIRECTION_ROW or
                  CUSPARSE_DIRECTION_COLUMN.                                    |
| `mb`          | number of block rows of sparse matrix $A$.                                  |
| `nb`          | number of block columns of sparse matrix $A$.                               |
| `descrA`      | the descriptor of matrix $A$. The supported matrix type is
                  CUSPARSE_MATRIX_TYPE_GENERAL.                                       |
Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsrValA</td>
<td>`&lt;type&gt; array of nnzb<em>rowBlockDim</em>colBlockDim non-zero elements of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of mb+1 elements that contains the start of every block row and the end of the last block row plus one of matrix A.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb column indices of the non-zero blocks of matrix A.</td>
</tr>
<tr>
<td>rowBlockDim</td>
<td>number of rows within a block of A.</td>
</tr>
<tr>
<td>colBlockDim</td>
<td>number of columns within a block of A.</td>
</tr>
<tr>
<td>descrC</td>
<td>the descriptor of matrix C. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValC</td>
<td>`&lt;type&gt; array of nnz non-zero elements of matrix C.</td>
</tr>
<tr>
<td>csrRowPtrC</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one of matrix C.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>integer array of nnz column indices of the non-zero elements of matrix C.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 13.5. cusparse<t>csr2gebsr()
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseCcsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int m,
int n,
const cusparseMatDescr_t descrA,
const cuComplex* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseZcsr2gebsr_bufferSize(cusparseHandle_t handle,
cusparseDirection_t dir,
int m,
int n,
const cusparseMatDescr_t descrA,
const cuDoubleComplex* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
int rowBlockDim,
int colBlockDim,
int* pBufferSize)

cusparseStatus_t
cusparseXcsr2gebsrNnz(cusparseHandle_t handle,
cusparseDirection_t dir,
int m,
int n,
const cusparseMatDescr_t descrA,
const int* csrRowPtrA,
const int* csrColIndA,
const cusparseMatDescr_t descrC,
const int* bsrRowPtrC,
int rowBlockDim,
int colBlockDim,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseScsr2gebsr(cusparseHandle_t handle,
cusparseDirection_t dir,
int m,
int n,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const cusparseMatDescr_t descrC,
float* bsrValC,
int* bsrRowPtrC,
int* bsrColIndC,
This function converts a sparse matrix $A$ in CSR format (that is defined by arrays $csrValA$, $csrRowPtrA$, and $csrColIndA$) into a sparse matrix $C$ in general BSR format (that is defined by the three arrays $bsrValC$, $bsrRowPtrC$, and $bsrColIndC$).
The matrix $A$ is an $m \times n$ sparse matrix and matrix $C$ is a $(mb \times rowBlockDim) \times (nb \times colBlockDim)$ sparse matrix, where $mb = (m + rowBlockDim - 1) / rowBlockDim$ is the number of block rows of $C$, and $nb = (n + colBlockDim - 1) / colBlockDim$ is the number of block columns of $C$.

The block of $C$ is of size $rowBlockDim \times colBlockDim$. If $m$ is not multiple of $rowBlockDim$ or $n$ is not multiple of $colBlockDim$, zeros are filled in.

The implementation adopts a two-step approach to do the conversion. First, the user allocates $bsrRowPtrC$ of $mb + 1$ elements and uses function `cusparseXcsr2gebsrNnz()` to determine the number of nonzero block columns per block row. Second, the user gathers $nnzb$ (number of nonzero block columns of matrix $C$) from either $(nnzb = *nnzTotalDevHostPtr)$ or $(nnzb = bsrRowPtrC[mb] - bsrRowPtrC[0])$ and allocates $bsrValC$ of $nnzb \times rowBlockDim \times colBlockDim$ elements and $bsrColIndC$ of $nnzb$ integers. Finally function `cusparse[S|D|C|Z]csr2gebsr()` is called to complete the conversion.

The user must obtain the size of the buffer required by `csr2gebsr()` by calling `csr2gebsr_bufferSize()`, allocate the buffer, and pass the buffer pointer to `csr2gebsr()`.

The general procedure is as follows:

```c
// Given CSR format (csrRowPtrA, csrColIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + rowBlockDim - 1) / rowBlockDim;
int nb = (n + colBlockDim - 1) / colBlockDim;
int bufferSize;
void *pBuffer;
cusparseScsr2gebsr_bufferSize(handle, dir, m, n,
    descrA, csrValA, csrRowPtrA, csrColIndA,
    rowBlockDim, colBlockDim,
    &bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) * (mb + 1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2gebsrNnz(handle, dir, m, n,
    descrA, csrRowPtrA, csrColIndA,
    descrC, bsrRowPtrC, rowBlockDim, colBlockDim,
    nnzTotalDevHostPtr,
    pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzb = *nnzTotalDevHostPtr;
} else{
    cudaMemcpy(&nnzb, bsrRowPtrC + mb, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDim * colBlockDim) * nnzb);
cusparseScsr2gebsr(handle, dir, m, n,
    descrA,
    csrValA, csrRowPtrA, csrColIndA,
    descrC,
    bsrValC, bsrRowPtrC, bsrColIndC,
    rowBlockDim, colBlockDim,
    pBuffer);
```

The routine `cusparseXcsr2gebsrNnz()` has the following properties:

- The routine requires no extra storage
- The routine does not support asynchronous execution
The routine does **not** support CUDA graph capture

The routine `cusparse<t>csr2gebsr()` has the following properties:

- The routine requires no extra storage if `pBuffer` != `NULL`
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>dir</code></td>
<td>storage format of blocks, either <code>CUSPARSE_DIRECTION_ROW</code> or <code>CUSPARSE_DIRECTION_COLUMN</code>.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows of sparse matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>number of columns of sparse matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>descrA</code></td>
<td>the descriptor of matrix <code>A</code>. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td><code>csrValA</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnz</code> nonzero elements of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>csrRowPtrA</code></td>
<td>integer array of <code>m+1</code> elements that contains the start of every row and the end of the last row plus one of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>csrColIndA</code></td>
<td>integer array of <code>nnz</code> column indices of the nonzero elements of matrix <code>A</code>.</td>
</tr>
<tr>
<td><code>descrC</code></td>
<td>the descriptor of matrix <code>C</code>. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td><code>rowBlockDim</code></td>
<td>number of rows within a block of <code>C</code>.</td>
</tr>
<tr>
<td><code>colBlockDim</code></td>
<td>number of columns within a block of <code>C</code>.</td>
</tr>
<tr>
<td><code>pBuffer</code></td>
<td>buffer allocated by the user, the size is return by <code>csr2gebsr_bufferSize()</code>.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bsrValC</code></td>
<td><code>&lt;type&gt;</code> array of <code>nnzb*rowBlockDim*colBlockDim</code> nonzero elements of matrix <code>C</code>.</td>
</tr>
<tr>
<td><code>bsrRowPtrC</code></td>
<td>integer array of <code>mb+1</code> elements that contains the start of every block row and the end of the last block row plus one of matrix <code>C</code>.</td>
</tr>
<tr>
<td><code>bsrColIndC</code></td>
<td>integer array of <code>nnzb</code> column indices of the nonzero blocks of matrix <code>C</code>.</td>
</tr>
</tbody>
</table>
nnzTotalDevHostPtr | total number of nonzero blocks of matrix C. Pointer nnzTotalDevHostPtr can point to a device memory or host memory.

See cusparseStatus_t for the description of the return status

13.6. cusparse<t>coo2csr()

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).

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooRowInd</td>
<td>integer array of nnz uncompressed row indices.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of non-zeros of the sparse matrix (that is also the length of array cooRowInd).</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>idxBase</td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

| csrRowPtr | integer array of m+1 elements that contains the start of every row and the end of the last row plus one. |

See cusparseStatus_t for the description of the return status
13.7.  cusparse<t>csc2dense()
[DEPRECATED]

[[DEPRECATED]] use cusparseSparse2Dense() instead. The routine will be removed in the next major release

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.

- The routine requires no extra storage
The routine supports asynchronous execution
The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>cscValA</td>
<td>&lt;type&gt; array of nnz(=cscColPtrA(m)−cscColPtrA(0)) nonzero elements of matrix A.</td>
</tr>
<tr>
<td>cscRowIndA</td>
<td>integer array of nnz(=cscColPtrA(m)−cscColPtrA(0)) row indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td>cscColPtrA</td>
<td>integer array of n+1 elements that contains the start of every row and the end of the last column plus one.</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of dense array A.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>array of dimensions (lda, n) that is filled in with the values of the sparse matrix.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.

### 13.8. cusparse<t>csr2bsr()

```cpp
#define CUSPARSE_CSR2BSR_NNZ(handle, dir, m, n, descrA, csrRowPtrA, csrColIndA, blockDim, descrC, bsrRowPtrC, nnzTotalDevHostPtr) cusparseXcsr2bsrNnz(handle, dir, m, n, descrA, csrRowPtrA, csrColIndA, blockDim, descrC, bsrRowPtrC, nnzTotalDevHostPtr)
#define CUSPARSE_CSR2BSR(handle, dir, m, n, descrA, descrC) cusparseScsr2bsr(handle, dir, m, n, descrA, descrC)
```
This function converts a sparse matrix in CSR format that is defined by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA` into a sparse matrix in BSR format that is defined by arrays `bsrValC`, `bsrRowPtrC`, and `bsrColIndC`.

A is an m*n sparse matrix. The BSR format of A has mb block rows, nb block columns, and nnzb nonzero blocks, where mb=((m+blockDim-1)/blockDim) and nb=(n+blockDim-1)/blockDim.
If \( m \) or \( n \) is not multiple of \( \text{blockDim} \), zeros are filled in.

The conversion in cuSPARSE entails a two-step approach. First, the user allocates \( \text{bsrRowPtrC} \) of \( mb+1 \) elements and uses function \text{cusparseXcsr2bsrNnz()} to determine the number of nonzero block columns per block row. Second, the user gathers \( \text{nnzb} \) (number of non-zero block columns of matrix \( C \)) from either \( \text{nnzb}=\text{nnzTotalDevHostPtr} \) or \( \text{nnzb}=\text{bsrRowPtrC}[mb]-\text{bsrRowPtrC}[0] \) and allocates \( \text{bsrValC} \) of \( \text{nnzb} \cdot \text{blockDim} \cdot \text{blockDim} \) elements and \( \text{bsrColIndC} \) of \( \text{nnzb} \) elements. Finally function \text{cusparse[S|D|C|Z]csr2bsr90} is called to complete the conversion.

The general procedure is as follows:

```c
// Given CSR format (csrRowPtrA, csrColIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2bsrNnz(handle, dir, m, n,
   descrA, csrRowPtrA, csrColIndA,
   blockDim,
   descrC, bsrRowPtrC,
   nnzTotalDevHostPtr);
if (NULL != nnzTotalDevHostPtr){
   nnzb = *nnzTotalDevHostPtr;
}else{
   cudaMemcpy(&nnzb, bsrRowPtrC+mb, sizeof(int), cudaMemcpyDeviceToHost);
   cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
   nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dir, m, n,
   descrA,
   csrValA, csrRowPtrA, csrColIndA,
   blockDim,
   descrC,
   bsrValC, bsrRowPtrC, bsrColIndC);
```

The routine \text{cusparse<s|d|c|z>csr2bsr()} has the following properties:

- This function requires temporary extra storage that is allocated internally if \( \text{blockDim} > 16 \)
- The routine does not support asynchronous execution if \( \text{blockDim} == 1 \)
- The routine does not support CUDA graph capture if \( \text{blockDim} == 1 \)

The routine \text{cusparseXcsr2bsrNnz()} has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<p>| handle | handle to the cuSPARSE library context. |</p>
<table>
<thead>
<tr>
<th>dir</th>
<th>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>number of rows of sparse matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of sparse matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix $A$.</td>
</tr>
<tr>
<td>csrValA</td>
<td><code>&lt;type&gt;</code> array of $nnz = \text{csrRowPtrA}[m] - \text{csrRowPtrA}[0]$ non-zero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of $m+1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of $nnz$ column indices of the non-zero elements of matrix $A$.</td>
</tr>
<tr>
<td>blockDim</td>
<td>block dimension of sparse matrix $A$. The range of blockDim is between 1 and $\min(m,n)$.</td>
</tr>
<tr>
<td>descrC</td>
<td>the descriptor of matrix $C$.</td>
</tr>
</tbody>
</table>

**Output**

| bsrValC   | `<type>` array of $nnzb*\text{blockDim}*\text{blockDim}$ nonzero elements of matrix $C$. |
| bsrRowPtrC| integer array of $mb+1$ elements that contains the start of every block row and the end of the last block row plus one of matrix $C$. |
| bsrColIndC| integer array of $nnzb$ column indices of the non-zero blocks of matrix $C$. |
| nnzTotalDevHostPtr | total number of nonzero elements in device or host memory. It is equal to $(\text{bsrRowPtrC}[mb] - \text{bsrRowPtrC}[0])$. |

See [cusparseStatus_t](#) for the description of the return status

### 13.9. cusparse<t>csr2coo()

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).
The routine requires no extra storage
The routine supports asynchronous execution
The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>csrRowPtr</td>
<td>integer array of ( m+1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzeros of the sparse matrix (that is also the length of array cooRowInd).</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>idxBase</td>
<td>CUSPARSE_INDEX_BASE_ZERO OR CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooRowInd</td>
<td>integer array of ( nnz ) uncompressed row indices.</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

### 13.10. `cusparseCsr2cscEx2()`

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

```
cusparseStatus_t
cusparseCsr2cscEx2(cusparseHandle_t handle,
                  int m,
                  int n,
                  int nnz,
                  const void* csrVal,
                  const int* csrRowPtr,
                  const int* csrColInd,
                  void* cscVal,
                  int* cscColPtr,
                  int* cscRowInd,
```
This function converts a sparse matrix in CSR format (that is defined by the three arrays `csrVal`, `csrRowPtr`, and `csrColInd`) into a sparse matrix in CSC format (that is defined by arrays `cscVal`, `cscRowInd`, and `cscColPtr`). The resulting matrix can also be seen as the transpose of the original sparse matrix. Notice that this routine can also be used to convert a matrix in CSC format into a matrix in CSR format.

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

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

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

The function `cusparseCsr2cscEx2_bufferSize()` returns the size of the workspace needed by `cusparseCsr2cscEx2()`. User needs to allocate a buffer of this size and give that buffer to `cusparseCsr2cscEx2()` as an argument.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine does **not** support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>handle to the cuSPARSE library context</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows of the CSR input matrix; number of columns of the CSC output matrix</td>
</tr>
<tr>
<td><code>n</code></td>
<td>number of columns of the CSR input matrix; number of rows of the CSC output matrix</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>number of nonzero elements of the CSR and CSC matrices</td>
</tr>
<tr>
<td><code>csrVal</code></td>
<td>value array of size <code>nnz</code> of the CSR matrix; of same type as <code>valType</code></td>
</tr>
<tr>
<td><code>csrRowPtr</code></td>
<td>integer array of size <code>m + 1</code> that contains the CSR row offsets</td>
</tr>
<tr>
<td><code>csrColInd</code></td>
<td>integer array of size <code>nnz</code> that contains the CSR column indices</td>
</tr>
<tr>
<td><code>valType</code></td>
<td>value type for both CSR and CSC matrices</td>
</tr>
<tr>
<td><code>copyValues</code></td>
<td><code>CUSPARSE_ACTION_SYMBOLIC</code> or <code>CUSPARSE_ACTION_NUMERIC</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>Index base <code>CUSPARSE_INDEX_BASE_ZERO</code> or <code>CUSPARSE_INDEX_BASE_ONE</code></td>
</tr>
<tr>
<td>alg</td>
<td>algorithm implementation. see cusparseCsr2CscAlg_t for possible values.</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>bufferSize</td>
<td>number of bytes of workspace needed by cusparseCsr2cscEx2()</td>
</tr>
<tr>
<td>buffer</td>
<td>pointer to workspace buffer</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

13.11. cusparse<t>csr2dense()

[[DEPRECATED]] use cusparseSparse2Dense() instead. The routine will be removed in the next major release

```c
 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,
```
This function converts the sparse matrix in CSR format (that is defined by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \)) 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.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( m )</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>( n )</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is \text{CUSPARSE_MATRIX_TYPE_GENERAL}. Also, the supported index bases are \text{CUSPARSE_INDEX_BASE_ZERO} and \text{CUSPARSE_INDEX_BASE_ONE}.</td>
</tr>
<tr>
<td>( \text{csrValA} )</td>
<td>( &lt;\text{type}&gt; ) array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{csrRowPtrA} )</td>
<td>integer array of ( m+1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>( \text{csrColIndA} )</td>
<td>integer array of ( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) ) column indices of the nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{lda} )</td>
<td>leading dimension of array matrix ( A ).</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>array of dimensions ((\text{lda}, n)) that is filled in with the values of the sparse matrix.</td>
</tr>
</tbody>
</table>

See \text{cusparseStatus\_t} for the description of the return status

### 13.12. \text{cusparse\_t\_csr2csr\_compress()}
This function compresses the sparse matrix in CSR format into compressed CSR format. Given a sparse matrix A and a non-negative value threshold (in the case of complex values, only the magnitude of the real part is used in the check), the function returns a sparse matrix C, defined by

\[ C(i,j) = A(i,j) \quad \text{if} \quad |A(i,j)| > \text{threshold} \]

The implementation adopts a two-step approach to do the conversion. First, the user allocates csrRowPtrC of m+1 elements and uses function cusparse<t>nnz_compress() to
determine nnzPerRow (the number of nonzeros columns per row) and nnzC (the total number of nonzeros). Second, the user allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function cusparse<t>csr2csr_compress() is called to complete the conversion.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz} \left( = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) \right) ) elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>integer array of ( \text{nnz} \left( = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) \right) ) column indices of the elements of matrix ( A ).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>integer array of ( m+1 ) elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>nnzA</td>
<td>number of nonzero elements in matrix ( A ).</td>
</tr>
<tr>
<td>nnzPerRow</td>
<td>this array contains the number of elements kept in the compressed matrix, by row.</td>
</tr>
<tr>
<td>tol</td>
<td>on input, this contains the non-negative tolerance value used for compression. Any values in matrix ( A ) less than or equal to this value will be dropped during compression.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValC</td>
<td>on output, this array contains the typed values of elements kept in the compressed matrix. Size = nnzC.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>on output, this integer array contains the column indices of elements kept in the compressed matrix. Size = nnzC.</td>
</tr>
<tr>
<td>csrRowPtrC</td>
<td>on output, this integer array contains the row pointers for elements kept in the compressed matrix. Size = ( m+1 ).</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
The following is a sample code to show how to use this API.

```c
#include <stdio.h>
#include <sys/time.h>
#include <cusparse.h>

#define ERR_NE(X,Y) do { if ((X) != (Y)) { 
   fprintf(stderr,"Error in %s at %s:%d
",__func__,__FILE__,__LINE__); 
   exit(-1);}} while(0)
#define CUDA_CALL(X) ERR_NE((X),cudaSuccess)
#define CUSPARSE_CALL(X) ERR_NE((X),CUSPARSE_STATUS_SUCCESS)

int main(){
   int m = 6, n = 5;
   cusparseHandle_t handle;
   CUSPARSE_CALL(cusparseCreate(&handle));
   cusparseMatDescr_t descrX;
   CUSPARSE_CALL(cusparseCreateMatDescr(&descrX));
   // Initialize sparse matrix
   float *X;
   CUDA_CALL(cudaMallocManaged( &X, sizeof(float) * m * n ));
   memset( X, 0, sizeof(float) * m * n );
   X[0 + 0*m] = 1.0;  X[0 + 1*m] = 3.0;
   X[1 + 1*m] = -4.0; X[1 + 2*m] = 5.0;
   X[2 + 0*m] = 2.0;  X[2 + 3*m] = 7.0;  X[2 + 4*m] = 8.0;
   X[3 + 2*m] = 6.0;  X[3 + 4*m] = 9.0;
   X[4 + 3*m] = 3.5;  X[4 + 4*m] = 5.5;
   X[5 + 0*m] = 6.5;  X[5 + 2*m] = -9.9;
   // Initialize total_nnz, and nnzPerRowX for cusparseSdense2csr()
   int total_nnz = 13;
   int *nnzPerRowX;
   CUDA_CALL( cudaMallocManaged( &nnzPerRowX, sizeof(int) * m ));
   nnzPerRowX[0] = 2;  nnzPerRowX[1] = 2;  nnzPerRowX[2] = 3;
   float *csrValX;
   int *csrRowPtrX;
   int *csrColIndX;
   CUDA_CALL( cudaMallocManaged( &csrValX, sizeof(float) * total_nnz ));
   CUDA_CALL( cudaMallocManaged( &csrRowPtrX, sizeof(int) * (m+1)));
   CUDA_CALL( cudaMallocManaged( &csrColIndX, sizeof(int) * total_nnz ));

   Before calling this API, call two APIs to prepare the input.

   /** Call cusparseSdense2csr to generate CSR format as the inputs for 
   cusparseScsr2csr_compress **/
   CUSPARSE_CALL( cusparseSdense2csr( handle, m, n, descrX, X, 
   m, nnzPerRowX, csrValX, 
   csrRowPtrX, csrColIndX ));

   float tol = 3.5;
   int *nnzPerRowY;
   int *testNNZTotal;
   CUDA_CALL( cudaMallocManaged( &nnzPerRowY, sizeof(int) * m ));
   CUDA_CALL( cudaMallocManaged( &testNNZTotal, sizeof(int)));
   memset( nnzPerRowY, 0, sizeof(int) * m );
   // cusparseSnnz_compress generates nnzPerRowY and testNNZTotal
   CUSPARSE_CALL( cusparseSnnz_compress(handle, m, descrX, csrValX, 
   csrRowPtrX, nnzPerRowY, 
   testNNZTotal, tol));

   float *csrValY;
   int *csrRowPtrY;
   int *csrColIndY;
   CUDA_CALL( cudaMallocManaged( &csrValY, sizeof(float) * (*testNNZTotal)));
   CUDA_CALL( cudaMallocManaged( &csrRowPtrY, sizeof(int) * (m+1)));
   CUDA_CALL( cudaMallocManaged( &csrColIndY, sizeof(int) * (*testNNZTotal)));
```
CUSPARSE_CALL( cusparseScsr2csr_compress( handle, m, n, descrX, csrValX, 
csrColIndX, csrRowPtrX, 
total_nnz, nnzPerRowY, 
csrValY, csrColIndY, 
csrRowPtrY, tol));

/* Expect results */
nnzPerRowY:  0 2 2 2 1 2
csrValY:     -4 5 7 8 6 9 5.5 6.5 -9.9
csrColIndY:  1 2 3 4 2 4 4 0 2
csrRowPtrY:  0 0 2 4 6 7 9
*/
cudaFree(X);
cusparseDestroy(handle);
cudaFree(nnzPerRowX);
cudaFree(csrValX);
cudaFree(csrRowPtrX);
cudaFree(csrColIndX);
cudaFree(csrValY);
cudaFree(nnzPerRowY);
cudaFree(testNNZTotal);
cudaFree(csrRowPtrY);
cudaFree(csrColIndY);
return 0;
}

13.13. cusparse<t>dense2csc()

[DEPRECATED]

[[DEPRECATED]] use cusparseDense2Sparse() instead. The routine will be removed in the
next major release

cusparseStatus_t cusparseStatus_t
cusparseDdense2csc(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 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 cusparseStatus_t
cusparseDdense2csc(cusparseHandle_t handle, 
int m, 
int n,
const cusparseMatDescr_t descrA,
const float* A,
int lda,
const int* nnzPerCol,
float* cscValA,
int* cscRowIndA,
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 $\text{nnzPerCol}$, which can be precomputed with $\text{cusparse<t>nnz()}$.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix $A$. The supported matrix type is $\text{CUSPARSE_MATRIX_TYPE_GENERAL}$. Also, the supported index bases are $\text{CUSPARSE_INDEX_BASE_ZERO}$ and $\text{CUSPARSE_INDEX_BASE_ONE}$.</td>
</tr>
<tr>
<td>A</td>
<td>array of dimensions $(lda, n)$.</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of dense array $A$.</td>
</tr>
<tr>
<td>nnzPerCol</td>
<td>array of size $n$ containing the number of nonzero elements per column.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscValA</td>
<td>$\text{&lt;type&gt;}$ array of $\text{nnz(} = \text{cscRowPtrA}(m) - \text{cscRowPtrA}(0) \text{)}$ nonzero elements of matrix $A$. It is only filled in if $\text{copyValues}$ is set to $\text{CUSPARSE_ACTION_NUMERIC}$.</td>
</tr>
</tbody>
</table>
cscRowIndA: integer array of \( \text{nnz} (= \text{cscRowPtrA}(m) - \text{cscRowPtrA}(0)) \) row indices of the nonzero elements of matrix A.

cscColPtrA: integer array of \( n+1 \) elements that contains the start of every column and the end of the last column plus one.

See \texttt{cusparseStatus_t} for the description of the return status

### 13.14. \texttt{cusparse<t>}
\texttt{dense2csr()}

[DEPRECATED]

[[DEPRECATED]] use \texttt{cusparseDense2Sparse()} instead. The routine will be removed in the next major release.

```c
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 cuComplex* A, 
int lda, 
const int* nnzPerRow, 
cuComplex* 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 \( \text{nnzPerRow} \), which can be pre-computed with \( \text{cusparse<t>nnz()} \).

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

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( m )</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>( n )</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix ( A ). The supported matrix type is ( \text{CUSPARSE_MATRIX_TYPE_GENERAL} ). Also, the supported index bases are ( \text{CUSPARSE_INDEX_BASE_ZERO} ) and ( \text{CUSPARSE_INDEX_BASE_ONE} ).</td>
</tr>
<tr>
<td>( A )</td>
<td>array of dimensions ((\text{lda}, n)).</td>
</tr>
<tr>
<td>( \text{lda} )</td>
<td>leading dimension of dense array ( A ).</td>
</tr>
<tr>
<td>( \text{nnzPerRow} )</td>
<td>array of size ( n ) containing the number of non-zero elements per row.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{csrValA} )</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{csrRowPtrA} )</td>
<td>integer array of ( m+1 ) elements that contains the start of every column and the end of the last column plus one.</td>
</tr>
<tr>
<td>( \text{csrColIndA} )</td>
<td>integer array of ( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) column indices of the non-zero elements of matrix ( A ).</td>
</tr>
</tbody>
</table>

See \( \text{cusparseStatus\_t} \) for the description of the return status

### 13.15. \text{cusparse<t>nnz()}
This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dirA</td>
<td>direction that specifies whether to count nonzero elements by CUSPARSE_DIRECTION_ROW or by CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>A</td>
<td>matrix A.</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of A.</td>
</tr>
<tr>
<td>nnzPerRowColumn</td>
<td>number of nonzero elements per row or column.</td>
</tr>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>total number of nonzero elements in a dense matrix.</td>
</tr>
</tbody>
</table>
### n
- number of columns of matrix \( \mathbf{A} \).

### descrA
- the descriptor of matrix \( \mathbf{A} \). The supported matrix type is \texttt{CUSPARSE\_MATRIX\_TYPE\_GENERAL}. Also, the supported index bases are \texttt{CUSPARSE\_INDEX\_BASE\_ZERO} and \texttt{CUSPARSE\_INDEX\_BASE\_ONE}.

### A
- array of dimensions \((\text{lda}, n)\).

### lda
- leading dimension of dense array \( \mathbf{A} \).

#### Output
- \texttt{nnzPerRowColumn}
  - array of size \( m \) or \( n \) containing the number of nonzero elements per row or column, respectively.
- \texttt{nnzTotalDevHostPtr}
  - total number of nonzero elements in device or host memory.

See \texttt{cusparseStatus\_t} for the description of the return status.

### 13.16. \texttt{cusparseCreateIdentityPermutation()}

```c
int cusparseCreateIdentityPermutation(cusparseHandle_t handle, int n, int* p);
```

This function creates an identity map. The output parameter \( p \) represents such map by \( p = 0:1:(n-1) \).

This function is typically used with \texttt{coosort}, \texttt{csrsort}, \texttt{cscsort}.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

#### Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>size of the map.</td>
</tr>
</tbody>
</table>

#### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>device</td>
<td>integer array of dimensions ( n ).</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus\_t} for the description of the return status.
13.17. **cusparseXcoosort()**

```c
 cusparseStatus_t
cusparseXcoosort_bufferSizeExt(cusparseHandle_t handle,
                               int              m,
                               int              n,
                               int              nnz,
                               const int*       cooRows,
                               const int*       cooCols,
                               size_t*          pBufferSizeInBytes)
```

```c
 cusparseStatus_t
cusparseXcoosortByRow(cusparseHandle_t handle,
                      int              m,
                      int              n,
                      int              nnz,
                      int*             cooRows,
                      int*             cooCols,
                      int*             P,
                      void*            pBuffer)
```

```c
 cusparseStatus_t
cusparseXcoosortByColumn(cusparseHandle_t handle,
                        int              m,
                        int              n,
                        int              nnz,
                        int*             cooRows,
                        int*             cooCols,
                        int*             P,
                        void*            pBuffer);
```

This function sorts COO format. The sorting is in-place. Also the user can sort by row or sort by column.

A is an \( m \times n \) sparse matrix that is defined in COO storage format by the three arrays `cooVals`, `cooRows`, and `cooCols`.

There is no assumption for the base index of the matrix. `coosort` uses stable sort on signed integer, so the value of `cooRows` or `cooCols` can be negative.

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

The parameter `P` is both input and output. If the user wants to compute sorted `cooVal`, `P` must be set as `0:1:(nnz-1)` before `coosort()`, and after `coosort()`, new sorted value array satisfies `cooVal_sorted = cooVal(P)`.

Remark: the dimension `m` and `n` are not used. If the user does not know the value of `m` or `n`, just passes a value positive. This usually happens if the user only reads a COO array first and needs to decide the dimension `m` or `n` later.

Appendix section provides a simple example of `coosort()`.

- The routine requires no extra storage if `pBuffer != NULL`
The routine does **not** support asynchronous execution.

The routine does **not** support CUDA graph capture.

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>number of nonzero elements of matrix A.</td>
</tr>
<tr>
<td>cooRows</td>
<td>device</td>
<td>integer array of nnz unsorted row indices of A.</td>
</tr>
<tr>
<td>cooCols</td>
<td>device</td>
<td>integer array of nnz unsorted column indices of A.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>integer array of nnz unsorted map indices. To construct cooVal, the user has to set P=0:1:(nnz-1).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by coosort_bufferSizeExt().</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooRows</td>
<td>device</td>
<td>integer array of nnz sorted row indices of A.</td>
</tr>
<tr>
<td>cooCols</td>
<td>device</td>
<td>integer array of nnz sorted column indices of A.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>integer array of nnz sorted map indices.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See [ cusparseStatus_t](#) for the description of the return status.

13.18. **cusparseXcsrsort()**

```c
cusparseStatus_t cusparseXcsrsort_bufferSizeExt(cusparseHandle_t handle, int m, int n, int nnz, const int* csrRowPtr, const int* csrColInd, size_t* pBufferSizeInBytes)

cusparseStatus_t cusparseXcsrsort(cusparseHandle_t handle, int m, int n, int nnz, const cusparseMatDescr_t descrA, const int* csrRowPtr, const int* csrColInd, int* P, void* pBuffer)
```
This function sorts CSR format. The stable sorting is in-place.

The matrix type is regarded as CUSPARSE_MATRIX_TYPE_GENERAL implicitly. In other words, any symmetric property is ignored.

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

The parameter `P` is both input and output. If the user wants to compute sorted `csrVal`, `P` must be set as 0:1:nnz-1 before `csrsort()`, and after `csrsort()`, new sorted value array satisfies `csrVal_sorted = csrVal(P)`.

The general procedure is as follows:

```c
// A is a 3x3 sparse matrix, base-0
// | 1 2 3 |  
// | 4 5 6 |  
// | 7 8 9 | 
const int m = 3;
const int n = 3;
const int nnz = 9;
csrRowPtr[m+1] = { 0, 3, 6, 9}; // on device
csrColInd[nnz] = { 2, 1, 0, 0, 2,1, 1, 2, 0}; // on device
csrVal[nnz] = { 3, 2, 1, 4, 6, 5, 8, 9, 7}; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcsrsort_bufferSizeExt(handle, m, n, nnz, csrRowPtr, csrColInd,
                                &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSR format
cusparseXcsrsort(handle, m, n, nnz, descrA, csrRowPtr, csrColInd, P, pBuffer);

// step 4: gather sorted csrVal
cusparseDgthr(handle, nnz, csrVal, csrVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

- The routine requires no extra storage if `pBuffer` != NULL
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>number of nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowPtr</td>
<td>device</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
</tbody>
</table>
cuSPARSE Format Conversion Reference

cusparseXcscsort()

cusparseStatus_t
cusparseXcscsort_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    int nnz,
    const int* cscColPtr,
    const int* cscRowInd,
    size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseXcscsort(cusparseHandle_t handle,
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const int* cscColPtr,
    int* cscRowInd,
    int* P,
    void* pBuffer)

This function sorts CSC format. The stable sorting is in-place.

The matrix type is regarded as CUSPARSE_MATRIX_TYPE_GENERAL implicitly. In other words, any symmetric property is ignored.

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

The parameter P is both input and output. If the user wants to compute sorted cscVal, P must be set as 0:1:(nnz-1) before cscsort(), and after cscsort(), new sorted value array satisfies cscVal_sorted = cscVal(P).

The general procedure is as follows:

// A is a 3x3 sparse matrix, base-0
// A = | 1 2 |
// | 4 0 |
// | 0 8 |
const int m = 3;
const int n = 2;
const int nnz = 4;
cscColPtr[n+1] = { 0, 2, 4}; // on device
cscRowInd[nnz] = { 1, 0, 2, 0}; // on device
cscVal[nnz] = { 4.0, 1.0, 8.0, 2.0 }; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcscsort_bufferSizeExt(handle, m, n, nnz, cscColPtr, cscRowInd, &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSC format
cusparseXcscsort(handle, m, n, nnz, descrA, cscColPtr, cscRowInd, P, pBuffer);

// step 4: gather sorted cscVal
cusparseDgthr(handle, nnz, cscVal, cscVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);

- The routine requires no extra storage if pBuffer != NULL
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>number of nonzero elements of matrix A.</td>
</tr>
<tr>
<td>cscColPtr</td>
<td>device</td>
<td>integer array of n+1 elements that contains the start of every column and the end of the last column plus one.</td>
</tr>
<tr>
<td>cscRowInd</td>
<td>device</td>
<td>integer array of nnz unsorted row indices of A.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>integer array of nnz unsorted map indices. To construct cscVal, the user has to set P=0:1:(nnz-1).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by cscsort_bufferSizeExt().</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscRowInd</td>
<td>device</td>
<td>integer array of nnz sorted row indices of A.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>integer array of nnz sorted map indices.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>
See `cusparseStatus_t` for the description of the return status

### 13.20. cusparseXcsru2csr()

cusparseStatus_t
cusparseCreateCsru2csrInfo(csru2csrInfo_t *info);

cusparseStatus_t
cusparseDestroyCsru2csrInfo(csru2csrInfo_t info);

cusparseStatus_t
cusparseScsru2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnz,
float* csrVal,
const int* csrRowPtr,
int* csrColInd,
csru2csrInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsru2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnz,
double* csrVal,
const int* csrRowPtr,
int* csrColInd,
csru2csrInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsru2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnz,
cuComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csru2csrInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsru2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnz,
cuDoubleComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csru2csrInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseScsru2csr(cusparseHandle_t handle,
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
float* csrVal,
const int* csrRowPtr,
int* csrColInd,
const int* csrColInd,
int* csrColInd,
void* pBuffer)
cusparseStatus_t
cusparseDcsru2csr(cusparseHandle_t handle,
int m,
int n,
const cusparseMatDescr_t descrA,
double* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csrInfo_t info,
void* pBuffer)
cusparseStatus_t
cusparseCcsru2csr(cusparseHandle_t handle,
int m,
int n,
const cusparseMatDescr_t descrA,
cuComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csrInfo_t info,
void* pBuffer)
cusparseStatus_t
cusparseZcsru2csr(cusparseHandle_t handle,
int m,
int n,
const cusparseMatDescr_t descrA,
cuDoubleComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csrInfo_t info,
void* pBuffer)
cusparseStatus_t
cusparseScsr2csru(cusparseHandle_t handle,
int m,
int n,
const cusparseMatDescr_t descrA,
float* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csrInfo_t info,
void* pBuffer)
cusparseStatus_t
cusparseDcsr2csru(cusparseHandle_t handle,
int m,
int n,
int nnz,
This function transfers unsorted CSR format to CSR format, and vice versa. The operation is in-place.

This function is a wrapper of `csrsort` and `gthr`. The usecase is the following scenario.

If the user has a matrix $A$ of CSR format which is unsorted, and implements his own code (which can be CPU or GPU kernel) based on this special order (for example, diagonal first, then lower triangle, then upper triangle), and wants to convert it to CSR format when calling CUSPARSE library, and then convert it back when doing something else on his/her kernel.

For example, suppose the user wants to solve a linear system $Ax=b$ by the following iterative scheme

$$x^{(k+1)} = x^{(k)} + L^{-1} * (b - Ax^{(k)})$$

The code heavily uses SpMv and triangular solve. Assume that the user has an in-house design of SpMV (Sparse Matrix-Vector multiplication) based on special order of $A$. However the user wants to use CUSPARSE library for triangular solver. Then the following code can work.
do
step 1: compute residual vector \( r = b - A x^{(k)} \) by
step 2: \( B := \text{sort}(A) \), and \( L \) is lower triangular part of \( B \)
        (only sort \( A \) once and keep the permutation vector)
step 3: solve \( z = L^{(-1)} \ast (b) \)
step 4: add correction \( x^{(k+1)} = x^{(k)} + z \)
step 5: \( A := \text{unsort}(B) \)
(Use permutation vector to get back the unsorted CSR)

until convergence

The requirements of step 2 and step 5 are
1. In-place operation.
2. The permutation vector \( P \) is hidden in an opaque structure.
3. No \texttt{cudaMalloc} inside the conversion routine. Instead, the user has to provide the buffer explicitly.
4. The conversion between unsorted CSR and sorted CSR may needs several times, but the function only generates the permutation vector \( P \) once.
5. The function is based on \texttt{csrsort}, \texttt{gather} and \texttt{scatter} operations.

The operation is called \texttt{csru2csr}, which means unsorted CSR to sorted CSR. Also we provide the inverse operation, called \texttt{csr2csru}.

In order to keep the permutation vector invisible, we need an opaque structure called \texttt{csru2csrInfo}. Then two functions \texttt{cusparseCreateCsru2csrInfo}, \texttt{cusparseDestroyCsru2csrInfo} are used to initialize and to destroy the opaque structure.

\texttt{cusparse[S|D|C|Z]csru2csr\_bufferSizeExt} returns the size of the buffer. The permutation vector \( P \) is also allocated inside \texttt{csru2csrInfo}. The lifetime of the permutation vector is the same as the lifetime of \texttt{csru2csrInfo}.

\texttt{cusparse[S|D|C|Z]csru2csr} performs forward transformation from unsorted CSR to sorted CSR. First call uses \texttt{csrsort} to generate the permutation vector \( P \), and subsequent call uses \( P \) to do transformation.

\texttt{cusparse[S|D|C|Z]csr2csru} performs backward transformation from sorted CSR to unsorted CSR. \( P \) is used to get unsorted form back.

The routine \texttt{cusparse\_t\_csru2csr()} has the following properties:

- The routine requires no extra storage if \( \texttt{pBuffer} \neq \texttt{NULL} \)
- The routine does \textbf{not} support asynchronous execution
- The routine does \textbf{not} support CUDA graph capture

The routine \texttt{cusparse\_t\_csr2csru()} has the following properties if \( \texttt{pBuffer} \neq \texttt{NULL} \):

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture
The following tables describe parameters of `csr2csru_bufferSizeExt` and `csr2csru`.

### Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>nnz</td>
<td>host</td>
<td>number of nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td>csrVal</td>
<td>device</td>
<td>&lt;type&gt; array of nnz unsorted nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrRowsPtr</td>
<td>device</td>
<td>integer array of $m+1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColInd</td>
<td>device</td>
<td>integer array of nnz unsorted column indices of $A$.</td>
</tr>
<tr>
<td>info</td>
<td>host</td>
<td>opaque structure initialized using <code>cusparseCreateCsru2csrInfo()</code>.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by <code>csru2csr_bufferSizeExt()</code>.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrVal</td>
<td>device</td>
<td>&lt;type&gt; array of nnz sorted nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrColInd</td>
<td>device</td>
<td>integer array of nnz sorted column indices of $A$.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.

### 13.21. `cusparseXpruneDense2csr()`

```c

cusparseStatus_t
cusparseHpruneDense2csr_bufferSizeExt(cusparseHandle_t          handle,
                        int                   m,
                        int                   n,
                        const __half*            A,
                        int                   lda,
                        const __half*            threshold,
                        const cusparseMatDescr_t descrC,
                        const __half*            csrValC,
                        const int*               csrRowPtrC,
                        const int*               csrColIndC,
                        size_t*                  pBufferSizeInBytes)
```
cuSPARSE Format Conversion Reference

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusparseSpruneDense2csr_bufferSizeExt</td>
<td><code>cusparseStatus_t cusparseSpruneDense2csr_bufferSizeExt(cusparseHandle_t handle, int m, int n, const float* A, int lda, const float* threshold, const cusparseMatDescr_t descrC, const float* csrValC, const int* csrRowPtrC, const int* csrColIndC, size_t* pBufferSizeInBytes)</code></td>
</tr>
<tr>
<td>cusparseDpruneDense2csr_bufferSizeExt</td>
<td><code>cusparseStatus_t cusparseDpruneDense2csr_bufferSizeExt(cusparseHandle_t handle, int m, int n, const double* A, int lda, const double* threshold, const cusparseMatDescr_t descrC, const double* csrValC, const int* csrRowPtrC, const int* csrColIndC, size_t* pBufferSizeInBytes)</code></td>
</tr>
<tr>
<td>cusparseHpruneDense2csrNnz</td>
<td><code>cusparseStatus_t cusparseHpruneDense2csrNnz(cusparseHandle_t handle, int m, int n, const __half* A, int lda, const __half* threshold, const cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer)</code></td>
</tr>
<tr>
<td>cusparseSpruneDense2csrNnz</td>
<td><code>cusparseStatus_t cusparseSpruneDense2csrNnz(cusparseHandle_t handle, int m, int n, const float* A, int lda, const float* threshold, const cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer)</code></td>
</tr>
<tr>
<td>cusparseDpruneDense2csrNnz</td>
<td><code>cusparseStatus_t cusparseDpruneDense2csrNnz(cusparseHandle_t handle, int m, int n, const double* A, int lda, const double* threshold, const cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer)</code></td>
</tr>
</tbody>
</table>
This function prunes a dense matrix to a sparse matrix with CSR format.

Given a dense matrix $A$ and a non-negative value $\text{threshold}$, the function returns a sparse matrix $C$, defined by

$$ C(i,j) = A(i,j) \quad \text{if} \quad |A(i,j)| > \text{threshold} $$

The implementation adopts a two-step approach to do the conversion. First, the user allocates $\text{csrRowPtrC}$ of $m+1$ elements and uses function $\text{pruneDense2csrNnz()}$ to determine the number of nonzeros columns per row. Second, the user gathers $\text{nnzC}$ (number of nonzeros of matrix $C$) from either ($\text{nnzC}=*\text{nnzTotalDevHostPtr}$) or ($\text{nnzC}=\text{csrRowPtrC}[m]-\text{csrRowPtrC}[0]$) and allocates $\text{csrValC}$ of $\text{nnzC}$ elements and $\text{csrColIndC}$ of $\text{nnzC}$ integers. Finally function $\text{pruneDense2csr()}$ is called to complete the conversion.

The user must obtain the size of the buffer required by $\text{pruneDense2csr()}$ by calling $\text{pruneDense2csr_bufferSizeExt()}$, allocate the buffer, and pass the buffer pointer to $\text{pruneDense2csr()}$. 

<table>
<thead>
<tr>
<th>Type</th>
<th>cudsparseseatupsrparseHpruneDense2csr()</th>
</tr>
</thead>
<tbody>
<tr>
<td>int*</td>
<td></td>
</tr>
<tr>
<td>void*</td>
<td></td>
</tr>
<tr>
<td>nnzTotalDevHostPtr, pBuffer</td>
<td></td>
</tr>
</tbody>
</table>
Appendix section provides a simple example of `pruneDense2csr()`. The routine `cusparse<t>pruneDense2csrNnz()` has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

The routine `cusparse<t>DpruneDense2csr()` has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>array of dimension $(l_da, n)$.</td>
</tr>
<tr>
<td>lda</td>
<td>device</td>
<td>leading dimension of $A$. It must be at least max(1, m).</td>
</tr>
<tr>
<td>threshold</td>
<td>host or device</td>
<td>a value to drop the entries of $A$. threshold can point to a device memory or host memory.</td>
</tr>
<tr>
<td>descrC</td>
<td>host</td>
<td>the descriptor of matrix $C$. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported index bases are <code>CUSPARSE_INDEX_BASE_ZERO</code> and <code>CUSPARSE_INDEX_BASE_ONE</code>.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by <code>pruneDense2csr_bufferSizeExt()</code>.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>device or host</td>
<td>total number of nonzero of matrix $C$. nnzTotalDevHostPtr can point to a device memory or host memory.</td>
</tr>
<tr>
<td>csrValC</td>
<td>device</td>
<td>array of nnzC nonzero elements of matrix $C$.</td>
</tr>
<tr>
<td>csrRowsPtrC</td>
<td>device</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>device</td>
<td>integer array of nnzC column indices of $C$.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
13.22. cusparseXpruneCsr2csr()

cusparseStatus_t
cusparseXpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
double m,
double n,
double nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const double* threshold,
const cusparseMatDescr_t descrC,
const double* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
size_t*pBufferSizeInBytes)

cusparseStatus_t
cusparseSpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
float m,
float n,
float nnzA,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const float* threshold,
const cusparseMatDescr_t descrC,
const float* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
size_t*pBufferSizeInBytes)

cusparseStatus_t
cusparseDpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
double m,
double n,
double nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const double* threshold,
const cusparseMatDescr_t descrC,
const double* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
size_t*pBufferSizeInBytes)

cusparseStatus_t
cusparseHpruneCsr2csrNnz(cusparseHandle_t handle,
int m,
int n,
```c
int
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const __half* threshold,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseSpruneCsr2csrNnz(const handle, m, n,
int
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const float* threshold,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseDpruneCsr2csrNnz(const handle, m, n,
int
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const double* threshold,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseHpruneCsr2csr(const handle, m, n,
int
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const __half* threshold,
const cusparseMatDescr_t descrC,
__half* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
void* pBuffer)

cusparseStatus_t
cusparseSpruneCsr2csr(const handle, m, n,
int
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const __half* threshold,
const cusparseMatDescr_t descrC,
__half* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
void* pBuffer)
```
```c
int
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const float* threshold,
const cusparseMatDescr_t descrC,
float* csrValC,
const int* csrRowPtrC,
int* csrColIndC,
void* pBuffer)
cusparseStatus_t
cusparseDpruneCsr2csr(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const double* threshold,
const cusparseMatDescr_t descrC,
double* csrValC,
const int* csrRowPtrC,
int* csrColIndC,
void* pBuffer)
```

This function prunes a sparse matrix to a sparse matrix with CSR format.

Given a sparse matrix $A$ and a non-negative value $\text{threshold}$, the function returns a sparse matrix $C$, defined by

$$C_{i,j} = A_{i,j} \quad \text{if} \quad |A_{i,j}| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates `csrRowPtrC` of $m+1$ elements and uses function `pruneCsr2csrNnz()` to determine the number of nonzeros columns per row. Second, the user gathers `nnzC` (number of nonzeros of matrix $C$) from either `nnzC=*nnzTotalDevHostPtr` or `nnzC=csrRowPtrC[m]-csrRowPtrC[0])` and allocates `csrValC` of `nnzC` elements and `csrColIndC` of `nnzC` integers. Finally function `pruneCsr2csr()` is called to complete the conversion.

The user must obtain the size of the buffer required by `pruneCsr2csr()` by calling `pruneCsr2csr_bufferSizeExt()` , allocate the buffer, and pass the buffer pointer to `pruneCsr2csr()`.

Appendix section provides a simple example of `pruneCsr2csr()`.

The routine `cusparse<t>pruneCsr2csrNnz()` has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does **not** support asynchronous execution
- The routine does **not** support CUDA graph capture

The routine `cusparse<t>pruneCsr2csr()` has the following properties:

- The routine requires no extra storage
The routine supports asynchronous execution
The routine supports CUDA graph capture

Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix $A$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>&lt;type&gt; array of $nnzA$ nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>csrRowsPtrA</td>
<td>device</td>
<td>integer array of $m+1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>integer array of $nnzA$ column indices of $A$.</td>
</tr>
<tr>
<td>threshold</td>
<td>host or device</td>
<td>a value to drop the entries of $A$. threshold can point to a device memory or host memory.</td>
</tr>
<tr>
<td>descrC</td>
<td>host</td>
<td>the descriptor of matrix $C$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by pruneCsr2csr_bufferSizeExt().</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>device or host</td>
<td>total number of nonzero of matrix $C$. nnzTotalDevHostPtr can point to a device memory or host memory.</td>
</tr>
<tr>
<td>csrValC</td>
<td>device</td>
<td>&lt;type&gt; array of $nnzC$ nonzero elements of matrix $C$.</td>
</tr>
<tr>
<td>csrRowsPtrC</td>
<td>device</td>
<td>integer array of $m+1$ elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndC</td>
<td>device</td>
<td>integer array of $nnzC$ column indices of $C$.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

13.23. **cusparseXpruneDense2csrPercentage()**

cusparseStatus_t
cusparseHpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t handle, int m, int n, const __half* A, int lda, float percentage, const __half* descrC, const __half* csrValC, const int* csrRowPtrC, const int* csrColIndC, pruneInfo_t info, size_t* pBufferSizeInBytes)
cusparseSpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t handle, int m, int n, const float* A, int lda, float percentage, const cusparseMatDescr_t descrC, const float* csrValC, const int* csrRowPtrC, const int* csrColIndC, pruneInfo_t info, size_t* pBufferSizeInBytes)
cusparseDpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t handle, int m, int n, const double* A, int lda, double percentage, const cusparseMatDescr_t descrC, const double* csrValC, const int* csrRowPtrC, const int* csrColIndC, pruneInfo_t info, size_t* pBufferSizeInBytes)
int lda,
float percentage,
const cusparseMatDescr_t descrC,
const double* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
pruneInfo_t info,
size_t pBufferSizeInBytes)

cusparseStatus_t
cusparseHpruneDense2csrByPercentage(cusparseHandle_t handle,
int m,
int n,
const __half* A,
int lda,
float percentage,
const cusparseMatDescr_t descrC,
const cusparseMatDescr_t csrRowPtrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseSpruneDense2csrByPercentage(cusparseHandle_t handle,
int m,
int n,
const float* A,
int lda,
float percentage,
const cusparseMatDescr_t descrC,
const cusparseMatDescr_t csrRowPtrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseDpruneDense2csrByPercentage(cusparseHandle_t handle,
int m,
int n,
const double* A,
int lda,
float percentage,
const cusparseMatDescr_t descrC,
const cusparseMatDescr_t csrRowPtrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseHpruneDense2csrNnzByPercentage(cusparseHandle_t handle,
int m,
int n,
const __half* A,
int lda,
float percentage,
const cusparseMatDescr_t descrC,
const cusparseMatDescr_t csrRowPtrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)
This function prunes a dense matrix to a sparse matrix by percentage.

Given a dense matrix $A$ and a non-negative value $\text{percentage}$, the function computes sparse matrix $C$ by the following three steps:

**Step 1:** sort absolute value of $A$ in ascending order.

\[
\text{key} := \text{sort}( |A| )
\]

**Step 2:** choose threshold by the parameter $\text{percentage}$

\[
\begin{align*}
pos &= \text{ceil}(m \times n \times (\text{percentage}/100)) - 1 \\
p &= \text{min}(\text{pos}, m \times n - 1) \\
\text{threshold} &= \text{key}[\text{pos}]
\end{align*}
\]

**Step 3:** call $\text{pruneDense2csr()}$ by with the parameter $\text{threshold}$.

The implementation adopts a two-step approach to do the conversion. First, the user allocates $\text{csrRowPtrC}$ of $m+1$ elements and uses function $\text{pruneDense2csrNnzByPercentage()}$ to determine the number of nonzeros columns per row. Second, the user gathers
nnzC [number of nonzeros of matrix C] from either (nnzC=*nnzTotalDevHostPtr) or (nnzC=csrRowPtrC[m]-csrRowPtrC[0]) and allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function pruneDense2csrByPercentage() is called to complete the conversion.

The user must obtain the size of the buffer required by pruneDense2csrByPercentage() by calling pruneDense2csrByPercentage_bufferSizeExt(), allocate the buffer, and pass the buffer pointer to pruneDense2csrByPercentage().

Remark 1: the value of percentage must be not greater than 100. Otherwise, CUSPARSE_STATUS_INVALID_VALUE is returned.

Remark 2: the zeros of A are not ignored. All entries are sorted, including zeros. This is different from pruneCsr2csrByPercentage()

Appendix section provides a simple example of pruneDense2csrNnzByPercentage().

The routine cusparse<t>pruneDense2csrNnzByPercentage() has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

The routine cusparse<t>pruneDense2csrByPercentage() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>$A$</td>
<td>device</td>
<td>array of dimension $(lدا, n)$.</td>
</tr>
<tr>
<td>lda</td>
<td>device</td>
<td>leading dimension of $A$. It must be at least max(1, m).</td>
</tr>
<tr>
<td>percentage</td>
<td>host</td>
<td>percentage &lt;=100 and percentage &gt;= 0</td>
</tr>
<tr>
<td>descrC</td>
<td>host</td>
<td>the descriptor of matrix $C$. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by pruneDense2csrByPercentage_bufferSizeExt().</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>nnzTotalDevHostPtr</code></td>
<td>device or host</td>
<td>total number of nonzero of matrix C. <code>nnzTotalDevHostPtr</code> can point to a device memory or host memory.</td>
</tr>
<tr>
<td><code>csrValC</code></td>
<td>device</td>
<td><code>&lt;type&gt;</code> array of <code>nnzC</code> nonzero elements of matrix C.</td>
</tr>
<tr>
<td><code>csrRowsPtrC</code></td>
<td>device</td>
<td>integer array of <code>m+1</code> elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td><code>csrColIndC</code></td>
<td>device</td>
<td>integer array of <code>nnzC</code> column indices of C.</td>
</tr>
<tr>
<td><code>pBufferSizeInBytes</code></td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 13.24. cusparseXpruneCsr2csrByPercentage()

```c

cusparseStatus_t
cusparseXpruneCsr2csrByPercentage_bufferSizeExt (cusparseHandle_t handle,
                                          int m, int n,
                                          int nnzA,
                                          const cusparseMatDescr_t descrA,
                                          const __half* csrValA,
                                          const int* csrRowPtrA,
                                          const int* csrColIndA,
                                          float percentage,
                                          const cusparseMatDescr_t descrC,
                                          const __half* csrValC,
                                          const int* csrRowPtrC,
                                          const int* csrColIndC,
                                          pruneInfo_t info,
                                          size_t* pBufferSizeInBytes)
```

```c

cusparseStatus_t
cusparseSpruneCsr2csrByPercentage_bufferSizeExt (cusparseHandle_t handle,
                                          int m, int n,
                                          int nnzA,
                                          const cusparseMatDescr_t descrA,
                                          const float* csrValA,
```

---
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
const float* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
pruneInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseDpruneCsr2csrByPercentage_bufferSizeExt(cusparseHandle_t
handle,

int m,
int n,

int nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
const double* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
pruneInfo_t info,
size_t* pBufferSizeInBytes)

cusparseStatus_t
cusparseHpruneCsr2csrNnzByPercentage(cusparseHandle_t
handle,

int m,
int n,

int nnzA,
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
const int* csrValC,
const int* csrRowPtrC,
### cuSPARSE Format Conversion Reference

**cuSPARSE Library**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cusparseSpruneCsr2csrNnzByPercentage</code></td>
<td>Prune CSR to CSR matrix by percentage</td>
</tr>
<tr>
<td><code>cusparseDpruneCsr2csrNnzByPercentage</code></td>
<td>Prune CSR to CSR matrix by percentage (double precision)</td>
</tr>
<tr>
<td><code>cusparseHpruneCsr2csrByPercentage</code></td>
<td>Prune CSR to CSR matrix by percentage (half precision)</td>
</tr>
<tr>
<td><code>cusparseSpruneCsr2csrByPercentage</code></td>
<td>Prune CSR to CSR matrix by percentage (single precision)</td>
</tr>
</tbody>
</table>

#### cuSPARSE SpruneCSR To CSR Nnz By Percentage

```c

int*  nnzTotalDevHostPtr,
pruneInfo_t info,
void*  pBuffer)

cusparseStatus_t

cusparseSpruneCsr2csrNnzByPercentage(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

```

#### cuSPARSE DpruneCSR To CSR Nnz By Percentage

```c

int*  nnzTotalDevHostPtr,
pruneInfo_t info,
void*  pBuffer)

cusparseStatus_t

cusparseDpruneCsr2csrNnzByPercentage(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

```

#### cuSPARSE HpruneCSR To CSR By Percentage

```c

int*  nnzTotalDevHostPtr,
pruneInfo_t info,
void*  pBuffer)

cusparseStatus_t

cusparseHpruneCsr2csrByPercentage(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
__half* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
pruneInfo_t info,
void* pBuffer)

```

#### cuSPARSE SpruneCSR To CSR By Percentage

```c

int*  nnzTotalDevHostPtr,
pruneInfo_t info,
void*  pBuffer)

cusparseStatus_t

cusparseSpruneCsr2csrByPercentage(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
float percentage,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

```
This function prunes a sparse matrix to a sparse matrix by percentage.

Given a sparse matrix $A$ and a non-negative value $\text{percentage}$, the function computes sparse matrix $C$ by the following three steps:

Step 1: sort absolute value of $A$ in ascending order.

$$\text{key} := \text{sort}( |\text{csrValA}| )$$

Step 2: choose threshold by the parameter $\text{percentage}$

$$\text{pos} = \text{ceil}(\text{nnzA} \times (\text{percentage}/100)) - 1$$
$$\text{pos} = \text{min}(\text{pos}, \text{nnzA} - 1)$$
$$\text{pos} = \text{max}(\text{pos}, 0)$$
$$\text{threshold} = \text{key}[\text{pos}]$$

Step 3: call pruneCsr2csr() by with the parameter $\text{threshold}$.

The implementation adopts a two-step approach to do the conversion. First, the user allocates $\text{csrRowPtrC}$ of $m+1$ elements and uses function pruneCsr2csrNnzByPercentage() to determine the number of nonzeros columns per row. Second, the user gathers $\text{nnzC}$ [number of nonzeros of matrix C] from either ($\text{nnzC}=\text{nnzTotalDevHostPtr}$) or ($\text{nnzC}=\text{csrRowPtrC}[m]-\text{csrRowPtrC}[0]$) and allocates $\text{csrValC}$ of $\text{nnzC}$ elements and $\text{csrColIndC}$ of $\text{nnzC}$ integers. Finally function pruneCsr2csrByPercentage() is called to complete the conversion.

The user must obtain the size of the buffer required by pruneCsr2csrByPercentage() by calling pruneCsr2csrByPercentage_bufferSizeExt(), allocate the buffer, and pass the buffer pointer to pruneCsr2csrByPercentage().
Remark 1: the value of percentage must be not greater than 100. Otherwise, CUSPARSE_STATUS_INVALID_VALUE is returned.

Appendix section provides a simple example of pruneCsr2csrByPercentage().

The routine cusparse<t>pruneCsr2csrNnzByPercentage() has the following properties:
- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

The routine cusparse<t>pruneCsr2csrByPercentage() has the following properties:
- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>host</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>host</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix A.</td>
</tr>
<tr>
<td>nnzA</td>
<td>host</td>
<td>number of nonzeros of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>host</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>&lt;type&gt; array of nnzA nonzero elements of matrix A.</td>
</tr>
<tr>
<td>csrRowsPtrA</td>
<td>device</td>
<td>integer array of m+1 elements that contains the start of every row and the end of the last row plus one.</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>device</td>
<td>integer array of nnzA column indices of A.</td>
</tr>
<tr>
<td>percentage</td>
<td>host</td>
<td>percentage &lt;= 100 and percentage &gt;= 0</td>
</tr>
<tr>
<td>descrC</td>
<td>host</td>
<td>the descriptor of matrix C. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by pruneCsr2csrByPercentage_bufferSizeExt().</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>device or host</td>
<td>total number of nonzero of matrix C. nnzTotalDevHostPtr can point to a device memory or host memory.</td>
</tr>
<tr>
<td>csrValC</td>
<td>device</td>
<td>&lt;type&gt; array of nnzC nonzero elements of matrix C.</td>
</tr>
</tbody>
</table>
### 13.25. `cusparse<t>nnz_compress()`

This function is the step one to convert from CSR format to compressed CSR format.

Given a sparse matrix A and a non-negative value threshold, the function returns `nnzPerRow` (the number of nonzeros columns per row) and `nnzC` (the total number of nonzeros) of a sparse matrix C, defined by

$$C(i,j) = A(i,j) \text{ if } |A(i,j)| > \text{threshold}$$

- **csrRowsPtrC** device integer array of `m+1` elements that contains the start of every row and the end of the last row plus one.
- **csrColIndC** device integer array of `nnzC` column indices of C.
- **pBufferSizeInBytes** host number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status.
A key assumption for the cuComplex and cuDoubleComplex case is that this tolerance is given as the real part. For example tol = 1e-8 + 0*i and we extract cureal, that is the x component of this struct.

- This function requires temporary extra storage that is allocated internally
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix A.</td>
</tr>
<tr>
<td>descrA</td>
<td>the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>csrValA</td>
<td>csr noncompressed values array</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>the corresponding input noncompressed row pointer.</td>
</tr>
<tr>
<td>tol</td>
<td>non-negative tolerance to determine if a number less than or equal to it.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzPerRow</td>
<td>this array contains the number of elements whose absolute values are greater than tol per row.</td>
</tr>
<tr>
<td>nnzC</td>
<td>host/device pointer of the total number of elements whose absolute values are greater than tol.</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.
The cuSPARSE Generic APIs allow computing the most common sparse linear algebra operations, such as sparse matrix-vector (SpMV) and sparse matrix-matrix multiplication (SpMM), in a flexible way. The new APIs have the following capabilities and features:

- Set matrix data layouts, number of batches, and storage formats (for example, CSR, COO, and so on)
- Set input/output/compute data types. This also allows mixed data-type computation
- Set types of sparse matrix indices
- Choose the algorithm for the computation
- Provide external device memory for internal operations
- Provide extensive consistency checks across input matrices and vectors for a given routine. This includes the validation of matrix sizes, data types, layout, allowed operations, etc.

### 14.1. Generic Types Reference

The cuSPARSE generic type references are described in this section.

#### 14.1.1. cudaDataType_t

The section describes the types shared by multiple CUDA Libraries and defined in the header file `library_types.h`. The cudaDataType type is an enumerator to specify the data precision. It is used when the data reference does not carry the type itself (e.g. `void*`). For example, it is used in the routine `cusparseSpMM()`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Data Type</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td>The data type is 16-bit IEEE-754 floating-point</td>
<td>__half</td>
<td>cuda_fp16.h</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td>The data type is 16-bit complex IEEE-754 floating-point</td>
<td>__half2</td>
<td>cuda_fp16.h</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>The data type is 16-bit bfloat floating-point</td>
<td>__nv_bfloat16</td>
<td>cuda_bf16.h</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td>The data type is 16-bit complex bfloat floating-point</td>
<td>__nv_bfloat162</td>
<td>cuda_bf16.h</td>
</tr>
</tbody>
</table>
### cuSPARSE Generic API Reference

#### Value | Meaning | Data Type | Header |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>The data type is 32-bit IEEE-754 floating-point</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td>The data type is 32-bit complex IEEE-754 floating-point</td>
<td>cuComplex</td>
<td>cuComplex.h</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>The data type is 64-bit IEEE-754 floating-point</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td>The data type is 64-bit complex IEEE-754 floating-point</td>
<td>cuDoubleComplex</td>
<td>cuComplex.h</td>
</tr>
<tr>
<td>CUDA_R_8I</td>
<td>The data type is 8-bit integer</td>
<td>int8_t</td>
<td>stdint.h</td>
</tr>
<tr>
<td>CUDA_R_32I</td>
<td>The data type is 32-bit integer</td>
<td>int32_t</td>
<td>stdint.h</td>
</tr>
</tbody>
</table>

### IMPORTANT:
The Generic API routines allow all data types reported in the respective section of the documentation only on GPU architectures with *native* support for them. If a specific GPU model does not provide *native* support for a given data type, the routine returns CUSPARSE_STATUS_ARCH_MISMATCH error.

Unsupported data types and Compute Capability (CC):

- __half on GPUs with CC < 53 (e.g. Kepler)
- __nv_bfloat16 on GPUs with CC < 80 (e.g. Kepler, Maxwell, Pascal, Volta, Turing)

see https://developer.nvidia.com/cuda-gpus

#### 14.1.2. cusparseFormat_t

This type indicates the format of the sparse matrix.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_FORMAT_COO</td>
<td>The matrix is stored in Coordinate (COO) format organized in Structure of Arrays (SoA) layout</td>
</tr>
<tr>
<td>CUSPARSE_FORMAT_COO_AOS</td>
<td>The matrix is stored in Coordinate (COO) format organized in Array of Structures (SoA) layout</td>
</tr>
<tr>
<td>CUSPARSE_FORMAT_CSR</td>
<td>The matrix is stored in Compressed Sparse Row (CSR) format</td>
</tr>
</tbody>
</table>

#### 14.1.3. cusparseOrder_t

This type indicates the memory layout of a dense matrix. Currently, only column-major layout is supported.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_ORDER_ROW</td>
<td>The matrix is stored in row-major</td>
</tr>
<tr>
<td>CUSPARSE_ORDER_COL</td>
<td>The matrix is stored in column-major</td>
</tr>
</tbody>
</table>

#### 14.1.4. cusparseIndexType_t

This type indicates the index type for representing the sparse matrix indices.
### Value Meanings

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_INDEX_16U</td>
<td>16-bit unsigned integer [1, 65535]</td>
</tr>
<tr>
<td>CUSPARSE_INDEX_32I</td>
<td>32-bit signed integer [1, 2^31 - 1]</td>
</tr>
<tr>
<td>CUSPARSE_INDEX_64I</td>
<td>64-bit signed integer [1, 2^63 - 1]</td>
</tr>
</tbody>
</table>

### 14.2. Sparse Vector APIs

The cuSPARSE helper functions for sparse vector descriptor are described in this section.

#### 14.2.1. cusparseCreateSpVec()

```c
cusparseStatus_t
cusparseCreateSpVec(cusparseSpVecDescr_t* spVecDescr,
                     int64_t       size,
                     int64_t       nnz,
                     void*         indices,
                     void*         values,
                     cusparseIndexType_t idxType,
                     cusparseIndexBase_t idxBase,
                     cudaDataType  valueType)
```

This function initializes the sparse matrix descriptor `spVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spVecDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse vector descriptor</td>
</tr>
<tr>
<td>size</td>
<td>HOST</td>
<td>IN</td>
<td>Size of the sparse vector</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>IN</td>
<td>Number of non-zero entries of the sparse vector</td>
</tr>
<tr>
<td>indices</td>
<td>DEVICE</td>
<td>IN</td>
<td>Indices of the sparse vector. Array of size <code>nnz</code></td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse vector. Array of size <code>nnz</code></td>
</tr>
<tr>
<td>idxType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the data type of <code>indices</code></td>
</tr>
<tr>
<td>idxBase</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the the base index of <code>indices</code></td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of <code>values</code></td>
</tr>
</tbody>
</table>

**NOTE:** it is safe to cast away constness (`const_cast`) for input pointers if the descriptor will not be used as an output parameter of a routine (e.g. conversion functions).

See `cusparseStatus_t` for the description of the return status.

#### 14.2.2. cusparseDestroySpVec()

```c
cusparseStatus_t
cusparseDestroySpVec(cusparseSpVecDescr_t spVecDescr)
```

This function releases the host memory allocated for the sparse vector descriptor `spVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector descriptor</td>
</tr>
</tbody>
</table>
14.2.3. cusparseSpVecGet()

This function returns the fields of the sparse vector descriptor \texttt{spVecDescr}.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{spVecDescr}</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector descriptor</td>
</tr>
<tr>
<td>\texttt{size}</td>
<td>HOST</td>
<td>OUT</td>
<td>Size of the sparse vector</td>
</tr>
<tr>
<td>\texttt{nnz}</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse vector</td>
</tr>
<tr>
<td>\texttt{indices}</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Indices of the sparse vector. Array of size \texttt{nnz}</td>
</tr>
<tr>
<td>\texttt{values}</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse vector. Array of size \texttt{nnz}</td>
</tr>
<tr>
<td>\texttt{idxType}</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of \texttt{indices}</td>
</tr>
<tr>
<td>\texttt{idxBase}</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of \texttt{indices}</td>
</tr>
<tr>
<td>\texttt{valueType}</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of \texttt{values}</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus_t} for the description of the return status.

14.2.4. cusparseSpVecGetIndexBase()

This function returns the \texttt{idxBase} field of the sparse vector descriptor \texttt{spVecDescr}.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{spVecDescr}</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector descriptor</td>
</tr>
<tr>
<td>\texttt{idxBase}</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of \texttt{indices}</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus_t} for the description of the return status.

14.2.5. cusparseSpVecGetValues()

This function returns the \texttt{values} field of the sparse vector descriptor \texttt{spVecDescr}.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{spVecDescr}</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse vector descriptor</td>
</tr>
<tr>
<td>\texttt{values}</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of \texttt{indices}</td>
</tr>
</tbody>
</table>
### 14.2.6. cusparseSpVecSetValues()

```c
 cusparseStatus_t cusparseSpVecSetValues(cusparseSpVecDescr_t spVecDescr, void* values)
```

This function sets the `values` field of the sparse vector descriptor `spVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse vector. Array of size <code>nnz</code></td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

### 14.3. Sparse Matrix APIs

The cuSPARSE helper functions for sparse matrix descriptor are described in this section.

#### 14.3.1. cusparseCreateCoo()

```c
 cusparseStatus_t cusparseCreateCoo(cusparseSpMatDescr_t* spMatDescr, int64_t rows, int64_t cols, int64_t nnz, void* cooRowInd, void* cooColInd, void* cooValues, cusparseIndexType_t cooIdxType, cusparseIndexBase_t idxBase, cudaDataType valueType)
```

This function initializes the sparse matrix descriptor `spMatDescr` in the COO format (Structure of Arrays layout).

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>IN</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>IN</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>IN</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
<tr>
<td>cooRowInd</td>
<td>DEVICE</td>
<td>IN</td>
<td>Row indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td>cooColInd</td>
<td>DEVICE</td>
<td>IN</td>
<td>Column indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
</tbody>
</table>
### 14.3.2. `cusparseCreateCooAoS()`

```c
 cusparseStatus_t
 cusparseCreateCooAoS(cusparseSpMatDescr_t* spMatDescr,  
                     int64_t rows,                    
                     int64_t cols,                   
                     int64_t nnz,                   
                     void* cooInd,                  
                     void* cooValues,               
                     cusparseIndexType_t cooIdxType,  
                     cusparseIndexBase_t idxBase,    
                     cudaDataType valueType)
```

This function initializes the sparse matrix descriptor `spMatDescr` in the COO format (Array of Structures layout).

**Param.** | **Memory** | **In/out** | **Meaning** |
---|---|---|---|
`spMatDescr` | HOST | OUT | Sparse matrix descriptor |
`rows` | HOST | IN | Number of rows of the sparse matrix |
`cols` | HOST | IN | Number of columns of the sparse matrix |
`nnz` | HOST | IN | Number of non-zero entries of the sparse matrix |
`cooInd` | DEVICE | IN | \(<Row, Column>\) indices of the sparse matrix. Array of size `nnz` |
`cooValues` | DEVICE | IN | Values of the sparse matrix. Array of size `nnz` |
`cooIdxType` | HOST | IN | Enumerator specifying the data type of `cooInd` |
`idxBase` | HOST | IN | Enumerator specifying the base index of `cooInd` |
`valueType` | HOST | IN | Enumerator specifying the datatype of `cooValues` |

See `cusparseStatus_t` for the description of the return status.

### 14.3.3. `cusparseCreateCsr()`

```c
 cusparseStatus_t
 cusparseCreateCsr(cusparseSpMatDescr_t* spMatDescr,  
                   int64_t rows,  
                   int64_t cols,
```

...
This function initializes the sparse matrix descriptor `spMatDescr` in the CSR format.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spMatDescr</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>HOST</td>
<td>IN</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td><code>cols</code></td>
<td>HOST</td>
<td>IN</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>HOST</td>
<td>IN</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
<tr>
<td><code>csrRowOffsets</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Row offsets of the sparse matrix. Array of size <code>rows + 1</code></td>
</tr>
<tr>
<td><code>csrColInd</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Column indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>csrValues</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse matrix</td>
</tr>
<tr>
<td><code>csrRowOffsetsType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the data type of <code>csrRowOffsets</code></td>
</tr>
<tr>
<td><code>csrColIndType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the data type of <code>csrColInd</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the base index of <code>csrRowOffsets</code> and <code>csrColInd</code></td>
</tr>
<tr>
<td><code>valueType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of <code>csrValues</code></td>
</tr>
</tbody>
</table>

**NOTE:** it is safe to cast away constness (`const_cast`) for input pointers if the descriptor will not be used as an output parameter of a routine (e.g. conversion functions).

See `cusparseStatus_t` for the description of the return status

### 14.3.4. `cusparseCreateCsc()`

```c
int64_t nnz,
void* csrRowOffsets,
void* csrColInd,
void* csrValues,
cusparseIndexType_t csrRowOffsetsType,
cusparseIndexType_t csrColIndType,
cusparseIndexBase_t idxBase,
cudaDataType valueType)
```

This function initializes the sparse matrix descriptor `spMatDescr` in the CSC format.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spMatDescr</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>HOST</td>
<td>IN</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td><code>cols</code></td>
<td>HOST</td>
<td>IN</td>
<td>Number of columns of the sparse matrix</td>
</tr>
</tbody>
</table>
### 14.3.5. cusparseDestroySpMat()

```c
cusparse_status_t cusparseDestroySpMat(cusparseSpMatDesc_t spMatDescr)
```

This function releases the host memory allocated for the sparse matrix descriptor `spMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

### 14.3.6. cusparseCooGet()

```c
cusparse_status_t cusparseCooGet(const cusparseSpMatDesc_t spMatDescr, 
                                 int64_t* rows, 
                                 int64_t* cols, 
                                 int64_t* nnz, 
                                 void** cooRowInd, 
                                 void** cooColInd, 
                                 void** cooValues, 
                                 cusparseIndexType_t* idxType, 
                                 cusparseIndexBase_t* idxBase, 
                                 cudaDataType* valueType)
```

This function returns the fields of the sparse matrix descriptor `spMatDescr` stored in COO format [Array of Structures layout].

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td>Param.</td>
<td>Memory</td>
<td>In/out</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
<tr>
<td>coorInd</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Row indices of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>cooColInd</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Column indices of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>cooValues</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>cooIdxType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of coorInd and cooColInd</td>
</tr>
<tr>
<td>idxBase</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of coorInd and cooColInd</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of cooValues</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 14.3.7. cusparseCooAosGet()

```c

cusparseStatus_t cusparseCooAosGet(const cusparseSpMatDescr_t spMatDescr,
                                   int64_t* rows,
                                   int64_t* cols,
                                   int64_t* nnz,
                                   void** cooInd,
                                   void** cooValues,
                                   cusparseIndexType_t* idxType,
                                   cusparseIndexBase_t* idxBase,
                                   cudaDataType* valueType)
```

This function returns the fields of the sparse matrix descriptor `spMatDescr` stored in COO format [Structure of Arrays layout].

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
<tr>
<td>cooInd</td>
<td>DEVICE</td>
<td>OUT</td>
<td>&lt;Row, Column&gt; indices of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>cooValues</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>cooIdxType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of cooInd</td>
</tr>
<tr>
<td>idxBase</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of cooInd</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of cooValues</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 14.3.8. cusparseCsrGet()

```
cusparseStatus_t CUSPARSEAPI
```
This function returns the fields of the sparse matrix descriptor `spMatDescr` stored in CSR format.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td><code>cols</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
<tr>
<td><code>csrRowOffsets</code></td>
<td>DEVICE</td>
<td>OUT</td>
<td>Row offsets of the sparse matrix. Array of size <code>rows + 1</code></td>
</tr>
<tr>
<td><code>csrColInd</code></td>
<td>DEVICE</td>
<td>OUT</td>
<td>Column indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>csrValues</code></td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>csrRowOffsetsType</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of <code>csrRowOffsets</code></td>
</tr>
<tr>
<td><code>csrColIndType</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of <code>csrColInd</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of <code>csrRowOffsets</code> and <code>csrColInd</code></td>
</tr>
<tr>
<td><code>valueType</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of <code>csrValues</code></td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

### 14.3.9. `cusparseCsrSetPointers()`

This function sets the pointers of the sparse matrix descriptor `spMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>csrRowOffsets</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Row offsets of the sparse matrix. Array of size <code>rows + 1</code></td>
</tr>
<tr>
<td><code>csrColInd</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Column indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>csrValues</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
14.3.10. cusparseSpMatGetSize()

cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetSize(cusparseSpMatDescr_t spMatDescr,
    int64_t*             rows,
    int64_t*             cols,
    int64_t*             nnz);

This function returns the sizes of the sparse matrix spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the sparse matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the sparse matrix</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse matrix</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

14.3.11. cusparseSpMatGetFormat()

cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetFormat(const cusparseSpMatDescr_t spMatDescr,
    cusparseFormat_t*          format);

This function returns the format field of the sparse matrix descriptor spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>format</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the storage format of the sparse matrix</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

14.3.12. cusparseSpMatGetIndexBase()

cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetIndexBase(const cusparseSpMatDescr_t spMatDescr,
    cusparseIndexBase_t*       idxBase);

This function returns the idxBase field of the sparse matrix descriptor spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>idxBase</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of the sparse matrix</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status
14.3.13. cusparseSpMatGetValues()

```c
 cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetValues(cusparseSpMatDescr_t spMatDescr,
                         void** values)
```

This function returns the values field of the sparse matrix descriptor spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse matrix. Array of size nnz</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

14.3.14. cusparseSpMatSetValues()

```c
 cusparseStatus_t CUSPARSEAPI
cusparseSpMatSetValues(cusparseSpMatDescr_t spMatDescr,
                        void* values)
```

This function sets the values field of the sparse matrix descriptor spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse matrix. Array of size nnz</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

14.3.15. cusparseSpMatGetStridedBatch()

```c
 cusparseStatus_t
cusparseSpMatGetStridedBatch(const cusparseSpMatDescr_t spMatDescr,
                              int* batchCount)
```

This function returns the batchCount field of the sparse matrix descriptor spMatDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>batchCount</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of batches of the sparse matrix</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status
14.3.16. `cusparseSpMatSetStridedBatch()`  
**[DEPRECATED]**

[[DEPRECATED]] use `cusparseSpMatSetCsrStridedBatch()`, `cusparseSpMatSetCooStridedBatch()` instead. The routine will be removed in the next major release

```c

cusparseStatus_t

cusparseSpMatSetStridedBatch(cusparseSpMatDescr_t spMatDescr, int batchCount)
```

This function sets the `batchCount` field of the sparse matrix descriptor `spMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>batchCount</td>
<td>HOST</td>
<td>IN</td>
<td>Number of batches of the sparse matrix</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

14.3.17. `cusparseCooSetStridedBatch()`

```c

cusparseStatus_t

cusparseCooSetStridedBatch(cusparseSpMatDescr_t spMatDescr, int batchCount, int64_t batchStride)
```

This function sets the `batchCount` and the `batchStride` fields of the sparse matrix descriptor `spMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>batchCount</td>
<td>HOST</td>
<td>IN</td>
<td>Number of batches of the sparse matrix</td>
</tr>
<tr>
<td>batchStride</td>
<td>HOST</td>
<td>IN</td>
<td>address offset between consecutive batches</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

14.3.18. `cusparseCsrSetStridedBatch()`

```c

cusparseStatus_t

cusparseCsrSetStridedBatch(cusparseSpMatDescr_t spMatDescr, int batchCount, int64_t offsetsBatchStride, int64_t columnsValuesBatchStride)
```

This function sets the `batchCount` and the `batchStride` fields of the sparse matrix descriptor `spMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
</tbody>
</table>
### Dense Vector APIs

The cuSPARSE helper functions for dense vector descriptor are described in this section.

#### 14.4.1. cusparseCreateDnVec()

```c
cusparseStatus_t cusparseCreateDnVec(cusparseDnVecDescr_t* dnVecDescr,
                                    int64_t size,
                                    void** values,
                                    cudaDataType valueType)
```

This function initializes the dense vector descriptor `dnVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Dense vector descriptor</td>
</tr>
<tr>
<td>size</td>
<td>HOST</td>
<td>IN</td>
<td>Size of the dense vector</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense vector. Array of size <code>size</code></td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of <code>values</code></td>
</tr>
</tbody>
</table>

**NOTE:** it is safe to cast away constness (const_cast) for input pointers if the descriptor will not be used as an output parameter of a routine (e.g. conversion functions).

See `cusparseStatus_t` for the description of the return status.

#### 14.4.2. cusparseDestroyDnVec()

```c
cusparseStatus_t cusparseDestroyDnVec(cusparseDnVecDescr_t dnVecDescr)
```

This function releases the host memory allocated for the dense vector descriptor `dnVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector descriptor</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

#### 14.4.3. cusparseDnVecGet()

### 14.4. Dense Vector APIs

The cuSPARSE helper functions for dense vector descriptor are described in this section.

#### 14.4.1. cusparseCreateDnVec()

```c
cusparseStatus_t cusparseCreateDnVec(cusparseDnVecDescr_t* dnVecDescr,
                                    int64_t size,
                                    void** values,
                                    cudaDataType valueType)
```

This function initializes the dense vector descriptor `dnVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Dense vector descriptor</td>
</tr>
<tr>
<td>size</td>
<td>HOST</td>
<td>IN</td>
<td>Size of the dense vector</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense vector. Array of size <code>size</code></td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of <code>values</code></td>
</tr>
</tbody>
</table>

**NOTE:** it is safe to cast away constness (const_cast) for input pointers if the descriptor will not be used as an output parameter of a routine (e.g. conversion functions).

See `cusparseStatus_t` for the description of the return status.

#### 14.4.2. cusparseDestroyDnVec()

```c
cusparseStatus_t cusparseDestroyDnVec(cusparseDnVecDescr_t dnVecDescr)
```

This function releases the host memory allocated for the dense vector descriptor `dnVecDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector descriptor</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

#### 14.4.3. cusparseDnVecGet()
cusparseStatus_t
 cusparseDnVecGet(const cusparseDnVecDescr_t dnVecDescr,
               int64_t* size,
               void** values,
               cudaDataType* valueType)

This function returns the fields of the dense vector descriptor dnVecDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector descriptor</td>
</tr>
<tr>
<td>size</td>
<td>HOST</td>
<td>OUT</td>
<td>Size of the dense vector</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense vector. Array of size nnz</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of values</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

### 14.4.4. cusparseDnVecGetValues()

cusparseStatus_t
 cusparseDnVecGetValues(const cusparseDnVecDescr_t dnVecDescr,
                        void** values)

This function returns the values field of the dense vector descriptor dnVecDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense vector</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

### 14.4.5. cusparseDnVecSetValues()

cusparseStatus_t
 cusparseDnVecSetValues(cusparseDnVecDescr_t dnVecDescr,
                        void* values)

This function set the values field of the dense vector descriptor dnVecDescr.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnVecDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense vector. Array of size size</td>
</tr>
</tbody>
</table>

The possible error values returned by this function and their meanings are listed below :

See cusparseStatus_t for the description of the return status

### 14.5. Dense Matrix APIs

The cuSPARSE helper functions for dense matrix descriptor are described in this section.
### 14.5.1. `cusparseCreateDnMat()`

```c
cusparseStatus_t cusparseCreateDnMat(cusparseDnMatDescr_t* dnMatDescr,
    int64_t               rows,
    int64_t               cols,
    int64_t               ld,
    void*                 values,
    cudaDataType          valueType,
    cusparseOrder_t       order)
```

The function initializes the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnMatDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>IN</td>
<td>Number of rows of the dense matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>IN</td>
<td>Number of columns of the dense matrix</td>
</tr>
<tr>
<td>ld</td>
<td>HOST</td>
<td>IN</td>
<td>Leading dimension of the dense matrix</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense matrix. Array of size</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of</td>
</tr>
<tr>
<td>order</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the memory layout of</td>
</tr>
</tbody>
</table>

**NOTE:** it is safe to cast away constness (`const_cast`) for input pointers if the descriptor will not be used as an output parameter of a routine (e.g. conversion functions).

See `cusparseStatus_t` for the description of the return status.

### 14.5.2. `cusparseDestroyDnMat()`

```c
cusparseStatus_t cusparseDestroyDnMat(cusparseDnMatDescr_t dnMatDescr)
```

This function releases the host memory allocated for the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>dnMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.

### 14.5.3. `cusparseDnMatGet()`

```c
cusparseStatus_t cusparseDnMatGet(const cusparseDnMatDescr_t dnMatDescr,
    int64_t*                   rows,
    int64_t*                   cols,
    int64_t*                   ld,
    void**                     values,
    cudaDataType*              valueType,
    cusparseOrder_t            order)
```

The function initializes the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows</td>
<td>HOST</td>
<td>IN</td>
<td>Number of rows of the dense matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>IN</td>
<td>Number of columns of the dense matrix</td>
</tr>
<tr>
<td>ld</td>
<td>HOST</td>
<td>IN</td>
<td>Leading dimension of the dense matrix</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense matrix. Array of size</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of</td>
</tr>
<tr>
<td>order</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the memory layout of</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
This function returns the fields of the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dnMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td><code>rows</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the dense matrix</td>
</tr>
<tr>
<td><code>cols</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the dense matrix</td>
</tr>
<tr>
<td><code>ld</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Leading dimension of the dense matrix</td>
</tr>
<tr>
<td><code>values</code></td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense matrix. Array of size <code>ld * cols</code></td>
</tr>
<tr>
<td><code>valueType</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of <code>values</code></td>
</tr>
<tr>
<td><code>order</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the memory layout of the dense matrix</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 14.5.4. cusparseDnMatGetValues()

```c
 cusparseStatus_t CUSPARSEAPI
cusparseDnMatGetValues(const cusparseDnMatDescr_t dnMatDescr, 
                        void**                     values)
```

This function returns the `values` field of the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dnMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td><code>values</code></td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense matrix. Array of size <code>ld * cols</code></td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 14.5.5. cusparseDnSetValues()

```c
 cusparseStatus_t CUSPARSEAPI
cusparseDnMatSetValues(cusparseDnMatDescr_t dnMatDescr, 
                       void*                values)
```

This function sets the `values` field of the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dnMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td><code>values</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the dense matrix. Array of size <code>ld * cols</code></td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status

### 14.5.6. cusparseDnMatGetStridedBatch()

```c
 cusparseStatus_t
cusparseDnMatGetStridedBatch(const cusparseDnMatDescr_t dnMatDescr, 
                             int*                       batchCount, 
                             int64_t*                   batchStride)
```

This function returns the strided batch of the dense matrix descriptor `dnMatDescr`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dnMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td><code>batchCount</code></td>
<td>int</td>
<td></td>
<td>Number of batch elements</td>
</tr>
<tr>
<td><code>batchStride</code></td>
<td>int64_t</td>
<td></td>
<td>Batch stride</td>
</tr>
</tbody>
</table>
The function returns the number of batches and the batch stride of the dense matrix descriptor \(dnMatDescr\).

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dnMatDescr)</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td>(batchCount)</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of batches of the dense matrix</td>
</tr>
<tr>
<td>(batchStride)</td>
<td>HOST</td>
<td>OUT</td>
<td>Address offset between a matrix and the next one in the batch</td>
</tr>
</tbody>
</table>

See \(\text{cusparseStatus}_t\) for the description of the return status.

### 14.5.7. \(\text{cusparseDnMatSetStridedBatch}\)()

\[
\text{cusparseDnMatSetStridedBatch}(\text{cusparseDnMatDescr}_t \ dnMatDescr, \\
\text{int} \ \text{batchCount}, \\
\text{int64}_t \ \text{batchStride})
\]

The function sets the number of batches and the batch stride of the dense matrix descriptor \(dnMatDescr\).

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dnMatDescr)</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td>(batchCount)</td>
<td>HOST</td>
<td>IN</td>
<td>Number of batches of the dense matrix</td>
</tr>
<tr>
<td>(batchStride)</td>
<td>HOST</td>
<td>IN</td>
<td>Address offset between a matrix and the next one in the batch. (\text{batchStride} \geq \text{ld} \times \text{cols}) if the matrix uses column-major layout, (\text{batchStride} \geq \text{ld} \times \text{rows}) otherwise</td>
</tr>
</tbody>
</table>

See \(\text{cusparseStatus}_t\) for the description of the return status.

### 14.6. Generic API Functions

#### 14.6.1. \(\text{cusparseAxpby}\)()

\[
\text{cusparseAxpby}(\text{cusparseHandle}_t \ handle, \\
\text{const void}_t \ \alpha, \\
\text{const void}_t \ vecX, \\
\text{const void}_t \ \beta, \\
\text{cusparseDnVecDescr}_t \ vecY)
\]

The function computes the sum of a sparse vector \(vecX\) and a dense vector \(vecY\)

\[
Y = \alpha X + \beta Y
\]

In other words,

for \(i=0\) to \(\text{nnz}-1\)

\[
Y[X_{\text{indices}}[i]] = \alpha \times X_{\text{values}}[i] + \beta \times Y[X_{\text{indices}}[i]]
\]
### cusparseAxpby

**cusparseAxpby** supports the following index type for representing the sparse vector `vecX`:

- 32-bit indices ([CUSPARSE_INDEX_32I](#))
- 64-bit indices ([CUSPARSE_INDEX_64I](#))

**cusparseAxpby** supports the following datatypes:

<table>
<thead>
<tr>
<th>X/Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

**cusparseAxpby** has the following constraints:

- The arrays representing the sparse vector `vecX` must be aligned to 16 bytes

**cusparseAxpby** has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector `vecX` indices are distinct

**cusparseAxpby** supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See [cusparseStatus_t](#) for the description of the return status

Please visit [cuSPARSE Library Samples - cusparseAxpby](#) for a code example.
14.6.2. cusparseGather()

cusparseStatus_t
cusparseGather(cusparseHandle_t handle, 
cusparseDnVecDescr_t vecY, 
cusparseSpVecDescr_t vecX)

The function gathers the elements of the dense vector vecY into the sparse vector vecX in other words,

for i=0 to nnz-1
    X_values[i] = Y[X_indices[i]]

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse vector X</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector Y</td>
</tr>
</tbody>
</table>

cusparseGather supports the following index type for representing the sparse vector vecX:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)

cusparseGather supports the following datatypes:

<table>
<thead>
<tr>
<th>x/y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseGather() has the following constraints:

- The arrays representing the sparse vector vecX must be aligned to 16 bytes

cusparseGather() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector vecX indices are distinct

cusparseGather() supports the following optimizations:
CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseGather for a code example.

### 14.6.3. cusparseScatter()

```c
cusparseStatus_t cusparseScatter(cusparseHandle_t handle,
                                cusparseSpVecDescr_t vecX,
                                cusparseDnVecDescr_t vecY)
```

The function scatters the elements of the sparse vector `vecX` into the dense vector `vecY`

In other words,

```c
for i=0 to nnz-1
    Y[X_indices[i]] = X_values[i]
```

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector X</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>OUT</td>
<td>Dense vector Y</td>
</tr>
</tbody>
</table>

`cusparseScatter` supports the following index type for representing the sparse vector `vecX`:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)

`cusparseScatter` supports the following datatypes:

<table>
<thead>
<tr>
<th>x/y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

`cusparseScatter()` has the following constraints:

- The arrays representing the sparse vector `vecX` must be aligned to 16 bytes

`cusparseScatter()` has the following properties:
The routine requires no extra storage

The routine supports asynchronous execution

Provides deterministic (bit-wise) results for each run if the the sparse vector $\text{vecX}$ indices are distinct

cusparseScatter() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See cusparseStatus_t for the description of the return status

Please visit cuSPARSE Library Samples - cusparseScatter for a code example.

### 14.6.4. cusparseRot()

cusparseStatus_t cusparseRot(cusparseHandle_t handle, const void* c_coeff, const void* s_coeff, cusparseSpVecDescr_t vecX, cusparseDnVecDescr_t vecY)

The function computes the Givens rotation matrix

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

to a sparse $\text{vecX}$ and a dense vector $\text{vecY}$

In other words,

for $i=0$ to $\text{nnz}-1$

\[
\text{Y}[\text{X_indices}[i]] = c \cdot \text{Y}[\text{X_indices}[i]] - s \cdot \text{X_values}[i] \\
\text{X_values}[i] = c \cdot \text{X_values}[i] + s \cdot \text{Y}[\text{X_indices}[i]]
\]

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>c_coeff</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>cosine element of the rotation matrix</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Sparse vector $X$</td>
</tr>
<tr>
<td>s_coeff</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>sine element of the rotation matrix</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense vector $Y$</td>
</tr>
</tbody>
</table>

cusparseRot supports the following index type for representing the sparse vector $\text{vecX}$:

- 32-bit indices [CUSPARSE_INDEX_32I]
- 64-bit indices [CUSPARSE_INDEX_64I]

cusparseRot supports the following datatypes:
cuSPARSE Generic API Reference

<table>
<thead>
<tr>
<th>$x/y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseRot() has the following constraints:

- The arrays representing the sparse vector $\text{vecX}$ must be aligned to 16 bytes

cusparseRot() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector $\text{vecX}$ indices are distinct

cusparseRot() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See cusparseStatus_t for the description of the return status

Please visit cuSPARSE Library Samples - cusparseRot for a code example.

14.6.5. cusparseSpVV()
The function computes the inner dot product of a sparse vector \( \text{vecX} \) and a dense vector \( \text{vecY} \)

\[
\text{result} = \mathbf{X}^\top \cdot \mathbf{Y}
\]

In other words,

\[
\text{result} = 0; \\
\text{for } i=0 \text{ to } \text{nnz}-1 \\
\quad \text{result} += X_\text{values}[i] \times Y[X_\text{indices}[i]]
\]

\[
\text{op}(X) = \begin{cases} X & \text{if op}(X) = \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ \overline{X} & \text{if op}(X) = \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}
\]

The function \text{cusparseSpVV\_bufferSize()} returns the size of the workspace needed by \text{cusparseSpVV()}.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>opX</td>
<td>HOST</td>
<td>IN</td>
<td>Operation (\text{op}(X)) that is non-transpose or conjugate transpose</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector (X)</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector (Y)</td>
</tr>
<tr>
<td>result</td>
<td>HOST or DEVICE</td>
<td>OUT</td>
<td>The resulting dot product</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype in which the computation is executed</td>
</tr>
<tr>
<td>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by \text{cusparseSpVV()}</td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer</td>
</tr>
</tbody>
</table>

cusparseSpVV supports the following index type for representing the sparse vector \( \text{vecX} \):

- 32-bit indices \{CUSPARSE\_INDEX\_32I\}
- 64-bit indices \{CUSPARSE\_INDEX\_64I\}

The datatypes combinations currently supported for \text{cusparseSpVV} are listed below:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>X/Y/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>
Mixed-precision computation:

<table>
<thead>
<tr>
<th>X/Y</th>
<th>computeType/result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_32I</td>
</tr>
<tr>
<td>CUDA_R_8I</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td></td>
</tr>
</tbody>
</table>

cusparseSpVV() has the following constraints:
- The arrays representing the sparse vector vecX must be aligned to 16 bytes

cusparseSpVV() has the following properties:
- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector vecX indices are distinct

cusparseSpVV() supports the following optimizations:
- CUDA graph capture
- Hardware Memory Compression

See cusparseStatus_t for the description of the return status
Please visit cuSPARSE Library Samples - cusparseSpVV for a code example.

14.6.6. cusparseSparseToDense()

The function converts the sparse matrix matA in CSR, CSC, or COO format into its dense representation matB.

The function cusparseSparseToDense_bufferSize() returns the size of the workspace needed by cusparseSparseToDense()
### cuSparseSparseToDense() Parameters

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix A</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>OUT</td>
<td>Dense matrix B</td>
</tr>
<tr>
<td>alg</td>
<td>HOST</td>
<td>IN</td>
<td>Algorithm for the computation</td>
</tr>
<tr>
<td>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by cuSparseSparseToDense()</td>
</tr>
<tr>
<td>buffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer</td>
</tr>
</tbody>
</table>

### Index Types

- 32-bit indices ([CUSPARSE_INDEX_32I](#))
- 64-bit indices ([CUSPARSE_INDEX_64I](#))

### Datatypes

<table>
<thead>
<tr>
<th>A/B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td></td>
</tr>
</tbody>
</table>

### Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SPARSETODENSE_ALG_DEFAULT</td>
<td>Default algorithm</td>
</tr>
</tbody>
</table>

### Properties

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run

### Optimizations

- CUDA graph capture
- Hardware Memory Compression

See [cusparseStatus_t](#) for the description of the return status.

Please visit [cuSPARSE Library Samples - cuSparseSparseToDense](#) for a code example.
14.6.7. cusparseDenseToSparse()

The function converts the dense matrix $\text{matA}$ into a sparse matrix $\text{matB}$ in CSR, CSC, or COO format.

The function `cusparseDenseToSparse_bufferSize()` returns the size of the workspace needed by `cusparseDenseToSparse_analysis()`.

The function `cusparseDenseToSparse_analysis()` updates the number of non-zero elements in the sparse matrix descriptor $\text{matB}$. The user is responsible to allocate the memory required by the sparse matrix array:

- Row/Column indices and value arrays for CSC and CSR respectively
- Row, column, value arrays for COO

Finally, we call `cusparseDenseToSparse_convert()` for filling the arrays allocated in the previous step.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix $\text{A}$</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse matrix $\text{B}$</td>
</tr>
<tr>
<td>alg</td>
<td>HOST</td>
<td>IN</td>
<td>Algorithm for the computation</td>
</tr>
<tr>
<td>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by <code>cusparseDenseToSparse_analysis()</code></td>
</tr>
<tr>
<td>buffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer</td>
</tr>
</tbody>
</table>

cusparseDenseToSparse() supports the following index type for representing the sparse vector $\text{matB}$:
32-bit indices [CUSPARSE_INDEX_32I]
64-bit indices [CUSPARSE_INDEX_64I]

cusparseDenseToSparse() supports the following datatypes:

<table>
<thead>
<tr>
<th>A/B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseDense2Sparse() supports the following algorithm:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_DENSETOSPARSE_ALG_DEFAULT</td>
<td>Default algorithm</td>
</tr>
</tbody>
</table>

cusparseDenseToSparse() has the following properties:
- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run

cusparseDenseToSparse() supports the following optimizations:
- The routine supports does **not** support CUDA graph capture
- Hardware Memory Compression

See cusparseStatus_t for the description of the return status

Please visit cuSPARSE Library Samples - cusparseDenseToSparse for a code example.

14.6.8. cusparseSpMV()

cusparseStatus_t

cusparseSpMV_bufferSize(cusparseHandle_t handle,
                        cusparseOperation_t opA,
                        const void* alpha,
                        const cusparseSpMatDescr_t matA,
                        const cusparseDnVecDescr_t vecX,
                        const void* beta,
                        const cusparseDnVecDescr_t vecY,
                        cudaDataType computeType,
                        cusparseSpMVAlg_t alg)
This function performs the multiplication of a sparse matrix `matA` and a dense vector `vecX` where

\[ Y = \alpha op(A) \cdot X + \beta Y \]

where

- `op(A)` is a sparse matrix of size \( m \times k \)
- `X` is a dense vector of size \( k \)
- `Y` is a dense vector of size \( m \)
- `\alpha` and `\beta` are scalars

Also, for matrix `A`

\[
\begin{align*}
\text{op}(A) &= \begin{cases} 
A & \text{if } \text{op}(A) = \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\
A^T & \text{if } \text{op}(A) = \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\
A^H & \text{if } \text{op}(A) = \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE}
\end{cases}
\end{align*}
\]

When using the (conjugate) transpose of the sparse matrix `A`, this routine may produce slightly different results during different runs with the same input parameters.

The function `cusparseSpMV_bufferSize()` returns the size of the workspace needed by `cusparseSpMV()`.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handle</code></td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td><code>opA</code></td>
<td>HOST</td>
<td>IN</td>
<td>Operation <code>op(A)</code></td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td><code>\alpha</code> scalar used for multiplication</td>
</tr>
<tr>
<td><code>matA</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix <code>A</code></td>
</tr>
<tr>
<td><code>vecX</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector <code>X</code></td>
</tr>
<tr>
<td><code>beta</code></td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td><code>\beta</code> scalar used for multiplication</td>
</tr>
<tr>
<td><code>vecY</code></td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense vector <code>Y</code></td>
</tr>
<tr>
<td><code>computeType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype in which the computation is executed</td>
</tr>
</tbody>
</table>
### Param. | Memory | In/out | Meaning
--- | --- | --- | ---
alg | HOST | IN | Enumerator specifying the algorithm for the computation
bufferSize | HOST | OUT | Number of bytes of workspace needed by cusparseSpMV
externalBuffer | DEVICE | IN | Pointer to workspace buffer

cusparseSpMV supports the following index type for representing the sparse vector vecX:
- 32-bit indices [CUSPARSE_INDEX_32I]
- 64-bit indices [CUSPARSE_INDEX_64I]

cusparseSpMV supports the following datatypes:

#### Uniform-precision computation:

<table>
<thead>
<tr>
<th>A/X</th>
<th>computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td></td>
</tr>
</tbody>
</table>

Mixed-precision computation:

<table>
<thead>
<tr>
<th>A/X</th>
<th>Y</th>
<th>computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_32I</td>
<td>CUDA_R_32I</td>
</tr>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
</tr>
</tbody>
</table>

The sparse matrix formats currently supported are listed below:

<table>
<thead>
<tr>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_FORMAT_COO</td>
<td>May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_FORMAT_COO_AOS</td>
<td>May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_FORMAT_CSR</td>
<td>Provides deterministic [bit-wise] results for each run</td>
</tr>
</tbody>
</table>

cusparseSpMV supports the following algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_MV_ALG_DEFAULT</td>
<td>Default algorithm for any sparse matrix format</td>
</tr>
<tr>
<td>CUSPARSE_COOMV_ALG</td>
<td>Default algorithm for COO sparse matrix format</td>
</tr>
<tr>
<td>CUSPARSE_CSRMV_ALG1</td>
<td>Default algorithm for CSR sparse matrix format</td>
</tr>
</tbody>
</table>
The function has the following limitations:

- Half-precision is not supported with 64-bit indices (CUSPARSE_INDEX_64I)

`cusparseSpMV()` has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution

`cusparseSpMV()` supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status.

Please visit cuSPARSE Library Samples - cusparseSpMV CSR and cusparseSpMV COO for a code example.

14.6.9. `cusparseSpMM()`

```c
cusparseStatus_t
cusparseSpMM_bufferSize(cusparseHandle_t handle,
                          cusparseOperation_t opA,
                          cusparseOperation_t opB,
                          const void* alpha,
                          cusparseSpMatDescr_t matA,
                          cusparseDnMatDescr_t matB,
                          const void* beta,
                          cusparseDnMatDescr_t matC,
                          cudaDataType computeType,
                          cusparseSpMMAlg_t alg,
                          size_t* bufferSize)
```

```c
cusparseStatus_t
cusparseSpMM(cusparseHandle_t handle,
             cusparseOperation_t opA,
             cusparseOperation_t opB,
             const void* alpha,
             cusparseSpMatDescr_t matA,
             cusparseDnMatDescr_t matB,
             const void* beta,
             cusparseDnMatDescr_t matC,
             cudaDataType computeType,
             cusparseSpMMAlg_t alg,
             void* externalBuffer)
```

The function performs the multiplication of a sparse matrix `matA` and a dense matrix `matB`

\[ C = \alpha op(A) \cdot op(B) + \beta C \]
where

- $\text{op}(A)$ is a sparse matrix of size $m \times k$
- $\text{op}(B)$ is a dense matrix of size $k \times n$
- $C$ is a dense matrix of size $m \times n$
- $\alpha$ and $\beta$ are scalars

The routine can be also used to perform the multiplication of a dense matrix $\text{matB}$ and a sparse matrix $\text{matA}$ by switching the dense matrices layout:

$$
\begin{align*}
\text{C}_{C} & = B_{C} \cdot \text{A} + \beta C_{C} \\
\text{C}_{R} & = A^{T} \cdot B_{R} + \beta C_{R}
\end{align*}
$$

where $B_{C}$, $C_{C}$ indicate column-major layout, while $B_{R}$, $C_{R}$ refer to row-major layout.

Also, for matrix $A$ and $B$

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

$$
\begin{align*}
\text{op}(B) = \begin{cases} 
    B & \text{if op(B) == \text{CUSPARSE_\_OPERATION\_NON\_TRANSPOSE}} \\
    B^{T} & \text{if op(B) == \text{CUSPARSE_\_OPERATION\_TRANSPOSE}} \\
    B^{H} & \text{if op(B) == \text{CUSPARSE_\_OPERATION\_CONJUGATE\_TRANSPOSE}}
\end{cases}
\end{align*}
$$

When using the (conjugate) transpose of the sparse matrix $A$, this routine may produce slightly different results during different runs with the same input parameters.

The function `cusparseSpMM_bufferSize()` returns the size of the workspace needed by `cusparseSpMM()`

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>opA</td>
<td>HOST</td>
<td>IN</td>
<td>Operation $\text{op}(A)$</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\alpha$ scalar used for multiplication</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix $A$</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix $B$</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\beta$ scalar used for multiplication</td>
</tr>
<tr>
<td>matC</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense matrix $C$</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype in which the computation is executed</td>
</tr>
<tr>
<td>alg</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the algorithm for the computation</td>
</tr>
<tr>
<td>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by <code>cusparseSpMM</code></td>
</tr>
</tbody>
</table>
cusparseSpMM supports the following sparse matrix formats:

- CUSPARSE_FORMAT_COO
- CUSPARSE_FORMAT_CSR

cusparseSpMM supports the following index type for representing the sparse vector vecX:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I) only with CUSPARSE_SPMM_COO_ALG4 and CUSPARSE_SPMM_CSR_ALG2 algorithms

cusparseSpMM supports the following datatypes:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>A/B/ C/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA R_16F</td>
</tr>
<tr>
<td>CUDA R_16BF</td>
</tr>
<tr>
<td>CUDA R_32F</td>
</tr>
<tr>
<td>CUDA R_64F</td>
</tr>
<tr>
<td>CUDA C_16F</td>
</tr>
<tr>
<td>CUDA C_16BF</td>
</tr>
<tr>
<td>CUDA C_32F</td>
</tr>
<tr>
<td>CUDA C_64F</td>
</tr>
</tbody>
</table>

Mixed-precision computation:

<table>
<thead>
<tr>
<th>A/B</th>
<th>C</th>
<th>computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA R_8I</td>
<td>CUDA R_32I</td>
<td>CUDA R_32I</td>
</tr>
<tr>
<td>CUDA R_8I</td>
<td>CUDA R_16F</td>
<td>CUDA R_32F</td>
</tr>
<tr>
<td>CUDA R_16F</td>
<td>CUDA R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA R_16BF</td>
<td></td>
<td>CUDA R_32F</td>
</tr>
<tr>
<td>CUDA R_16F</td>
<td>CUDA_R_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA R_16BF</td>
<td></td>
<td>CUDA_R_16F</td>
</tr>
</tbody>
</table>

NOTE: CUDA R_16BF/CUDA C_16BF data types are supported only with CUSPARSE_SPMM_COO_ALG4 and CUSPARSE_SPMM_CSR_ALG2 algorithms

cusparseSpMM supports the following algorithms:

[D]: deprecated
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_MM_ALG_DEFAULT</td>
<td>Default algorithm for any sparse matrix format</td>
</tr>
<tr>
<td>CUSPARSE_SPMM_ALG_DEFAULT</td>
<td>Algorithm 1 for COO sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ May provide better performance for small number of nnz</td>
</tr>
<tr>
<td></td>
<td>‣ It supports only column-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ It supports batched computation</td>
</tr>
<tr>
<td></td>
<td>‣ May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_COOMM_ALG1</td>
<td>Algorithm 1 for COO sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ In general, slower than Algorithm 1 and 2</td>
</tr>
<tr>
<td></td>
<td>‣ It supports only column-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ It supports batched computation</td>
</tr>
<tr>
<td></td>
<td>‣ It provides deterministic result</td>
</tr>
<tr>
<td></td>
<td>‣ It requires additional memory</td>
</tr>
<tr>
<td>CUSPARSE_COOMM_ALG2</td>
<td>Algorithm 2 for COO sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ In general, slower than Algorithm 1 and 2</td>
</tr>
<tr>
<td></td>
<td>‣ It supports only column-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ It supports batched computation</td>
</tr>
<tr>
<td>CUSPARSE_COOMM_ALG3</td>
<td>Algorithm 3 for COO sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ May provide better performance for large number of nnz</td>
</tr>
<tr>
<td></td>
<td>‣ It supports only column-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_COOMM_ALG4</td>
<td>Algorithm 4 for COO sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ Provide the best performance with row-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ It supports batched computation</td>
</tr>
<tr>
<td></td>
<td>‣ May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_CSRMM_ALG1</td>
<td>Algorithm 1 for CSR sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ It provides deterministic result</td>
</tr>
<tr>
<td></td>
<td>‣ It supports only column-major layout</td>
</tr>
<tr>
<td>CUSPARSE_SPMM_CSR_ALG1</td>
<td>Algorithm 2 for CSR sparse matrix format</td>
</tr>
<tr>
<td></td>
<td>‣ Provide the best performance with row-major layout</td>
</tr>
<tr>
<td></td>
<td>‣ It supports batched computation</td>
</tr>
<tr>
<td></td>
<td>‣ May produce slightly different results during different runs with the same input parameters</td>
</tr>
</tbody>
</table>

**Performance notes:**

- Ensure grid sizes are not too small to avoid performance bottleneck.
- For best performance, use batched computation when available.
- Consider using different algorithms based on matrix properties and size.
- For large matrices, Algorithm 3 may provide better performance.
- Algorithm 4 provides best performance with row-major layout.

**Notes on Algorithms:**

- Algorithm 1 is generally faster for smaller matrices.
- Algorithm 3 is better for larger matrices.
- Algorithm 4 is optimized for row-major layout matrices.
- Algorithm 2 requires additional memory but provides deterministic results.
- Algorithm 4 requires additional memory for best performance.

---

**Algorithm Selection Tips:**

- Use CUSPARSE_MM_ALG_DEFAULT for general use.
- For COO format, use CUSPARSE_COOMM_ALG1 for small nnz, CUSPARSE_COOMM_ALG2 for general use, and CUSPARSE_COOMM_ALG3 for large nnz.
- For CSR format, use CUSPARSE_CSRMM_ALG1 for general use and CUSPARSE_SPMM_CSR_ALG2 for best row-major layout performance.

---

**References:**

- NVIDIA cuSPARSE Library
- cuSPARSE API Documentation

---

**License:**

- NVIDIA cuSPARSE Library
- GNU General Public License version 2 (GPLv2)

---

**Contact:**

- NVIDIA Technical Support
- NVIDIA Developer Forums

---

**Copyright:**

- © 2023 NVIDIA Corporation
- All rights reserved

---

**Disclaimer:**

- The information in this document is subject to change without notice.
- NVIDIA assumes no responsibility for any errors or omissions in the accuracy of the information contained herein.
- NVIDIA makes no commitment to update the information or release additional information in connection therewith.
- Row-major layout provides higher performance than column-major.
- CUSPARSE_SPMM_COO_ALG4 and CUSPARSE_SPMM_CSR_ALG2 should be used with row-major layout, while CUSPARSE_SPMM_COO_ALG1, CUSPARSE_SPMM_COO_ALG2, and CUSPARSE_SPMM_COO_ALG3, and CUSPARSE_SPMM_CSR_ALG1 with column-major layout.
- For beta != 1, the output matrix is scaled before the actual computation.

cusparseSpMM() with CUSPARSE_SPMM_COO_ALG4 and CUSPARSE_SPMM_CSR_ALG2 support the following batch modes:

- $C_i = A \cdot B_i$
- $C_i = A_i \cdot B$
- $C_i = A_i \cdot B_i$

The number of batches and their strides can be set by using cusparseCooSetStridedBatch, cusparseCsrSetStridedBatch, and cusparseDnMatSetStridedBatch.

cusparseSpMM() has the following properties:

- The routine requires no extra storage for CUSPARSE_SPMM_COO_ALG1, CUSPARSE_SPMM_COO_ALG3, CUSPARSE_SPMM_COO_ALG4, and CUSPARSE_SPMM_CSR_ALG1.
- The routine supports asynchronous execution.
- Provides deterministic (bit-wise) results for each run only for CUSPARSE_SPMM_COO_ALG2 and CUSPARSE_SPMM_CSR_ALG1 algorithms, and opA == CUSPARSE_OPERATION_NON_TRANSPOSE.

CUSPARSE_SPMM() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See cusparseStatus_t for the description of the return status.

Please visit cuSPARSE Library Samples - cusparseSpMM CSR and cusparseSpMM COO for a code example.

### 14.6.10. cusparseConstrainedGeMM()

cusparseStatus_t cusparseConstrainedGeMM(cusparseHandle_t handle, cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha, cusparseDnMatDescr_t matA, cusparseDnMatDescr_t matB, const void* beta, cusparseSpMatDescr_t matC, cudaDataType computeType, void* externalBuffer)

cusparseStatus_t cusparseConstrainedGeMM_bufferSize(cusparseHandle_t handle, cusparseOperation_t opA, cusparseOperation_t opB, const void* alpha, cusparseDnMatDescr_t matA, cusparseDnMatDescr_t matB, const void* beta, cusparseSpMatDescr_t matC, cudaDataType computeType, void* externalBuffer)
This function performs the multiplication of $\text{matA}$ and $\text{matB}$, followed by an element-wise multiplication with the sparsity pattern of $\text{matC}$. Formally, it performs the following operation:

$$ C = \alpha(\text{opA} \odot \text{opB}) \odot \text{spy}(C) + \beta C $$

where $\text{opA}$ is a dense matrix of size $m \times k$, $\text{opB}$ is a dense matrix of size $k \times n$, $C$ is a sparse matrix of size $m \times n$, $\alpha$ and $\beta$ are scalars, $\odot$ denotes the Hadamard (entry-wise) matrix product, and $\text{spy}(C)$ is the sparsity pattern matrix of $C$ defined as:

$$ \text{spy}(C)_{ij} = \begin{cases} 0 & \text{if } C_{ij} = 0 \\ 1 & \text{otherwise} \end{cases} $$

Matrices $\text{opA}$ and $\text{opB}$ are defined as:

$$ \text{opA} = \begin{cases} A & \text{if } \text{opA} = \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\ A^T & \text{if } \text{opA} = \text{CUSPARSE_OPERATION_TRANSPOSE} \\ A^H & \text{if } \text{opA} = \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE} \end{cases} $$

$$ \text{opB} = \begin{cases} B & \text{if } \text{opB} = \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \\ B^T & \text{if } \text{opB} = \text{CUSPARSE_OPERATION_TRANSPOSE} \\ B^H & \text{if } \text{opB} = \text{CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE} \end{cases} $$

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>opA</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the operation $\text{opA}$. Has to be $\text{CUSPARSE_OPERATION_NON_TRANSPOSE}$</td>
</tr>
<tr>
<td>opB</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the operation $\text{opB}$. Has to be $\text{CUSPARSE_OPERATION_NON_TRANSPOSE}$</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>Scalar $\alpha$ that scales the matrix product</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix $A$.</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix $B$.</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>Scalar $\beta$ that scales the accumulation matrix</td>
</tr>
<tr>
<td>matC</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Sparse matrix $C$.</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype used to execute the computation</td>
</tr>
<tr>
<td>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Size of $\text{externalBuffer}$ in bytes</td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least $\text{bufferSize}$ bytes</td>
</tr>
</tbody>
</table>
Currently, this function only supports \( \text{opA} = \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \) and \( \text{opB} = \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \). Attempting to pass a different operator will cause a \text{CUSPARSE\_STATUS\_NOT\_SUPPORTED} error.

The function has the following limitations:

- Only 32-bit indices \text{CUSPARSE\_INDEX\_32I} is supported.

The datatypes combinations currently supported for 
\text{cusparseSpMM} are listed below:

### Uniform-precision computation:

<table>
<thead>
<tr>
<th>( A/X/Y/\text{computeType} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

Currently supported sparse matrix formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_FORMAT_CSR</td>
<td>The column indices in each row must be sorted</td>
</tr>
</tbody>
</table>

\text{cusparseConstrainedGeMM()} has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution

\text{cusparseConstrainedGeMM()} supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See \text{cusparseStatus\_t} for the description of the return status.

### 14.6.11. \text{cusparseSpGEMM()} 

cusparseStatus\_t CUSPARSEAPI
\text{cusparseSpGEMM\_createDescr}(cusparseSpGEMM\_descr\_t* descr);
cusparseStatus\_t CUSPARSEAPI
\text{cusparseSpGEMM\_destroyDescr}(cusparseSpGEMM\_descr\_t descr);
cusparseStatus\_t CUSPARSEAPI
\text{cusparseSpGEMM\_workEstimation}(cusparse\_handle\_t handle, 
cusparse\_operation\_t operation\_t opA, 
cusparse\_operation\_t operation\_t opB, 
const void* alpha, 
cusparse\_sp\_mat\_descr\_t matA,}
This function performs the multiplication of two sparse matrices matA and matB

\[ C = \alpha \text{op}(A) \cdot \text{op}(B) + \beta C \]

where \( \alpha \) and \( \beta \) are scalars. Note that \( C \) and \( C \) must have the same sizes and the same sparsity pattern (for \( \beta \neq 0 \)).

The example \texttt{CSR SpGEMM()} shows the computation workflow for all steps.

The functions \texttt{cusparseSpGEMM_workEstimation()} and \texttt{cusparseSpGEMM_compute()} are used for both determining the buffer size and performing the actual computation.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>HOST</td>
<td>IN</td>
<td>Handle to the cuSPARSE library context</td>
</tr>
<tr>
<td>opA</td>
<td>HOST</td>
<td>IN</td>
<td>Operation ( \text{op}(A) )</td>
</tr>
<tr>
<td>opB</td>
<td>HOST</td>
<td>IN</td>
<td>Operation ( \text{op}(B) )</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>( \alpha ) scalar used for multiplication</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix ( A )</td>
</tr>
<tr>
<td>Param.</td>
<td>Memory</td>
<td>In/out</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>--------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix B</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\beta$ scalar used for multiplication</td>
</tr>
<tr>
<td>matC</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Sparse matrix C</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype in which the computation is executed</td>
</tr>
<tr>
<td>alg</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the algorithm for the computation</td>
</tr>
<tr>
<td>spgemmDescr</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Opaque descriptor for storing internal data used across the three steps</td>
</tr>
<tr>
<td>bufferSize1</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace needed by cusparseSpGEMM_workEstimation</td>
</tr>
<tr>
<td>bufferSize2</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace needed by cusparseSpGEMM_compute</td>
</tr>
<tr>
<td>externalBuffer1</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMM_workEstimation and cusparseSpGEMM_compute</td>
</tr>
<tr>
<td>externalBuffer2</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMM_compute and cusparseSpGEMM_copy</td>
</tr>
</tbody>
</table>

MEMORY REQUIREMENT: the first invocation of cusparseSpGEMM_compute provides an upper bound of the memory required for the computation that is generally several times larger of the actual memory used. The user can provide an arbitrary buffer size bufferSize2 in the second invocation. If it is not sufficient, the routine will returns CUSPARSE_STATUS_INSUFFICIENT_RESOURCES status.

Currently, the function has the following limitations:

- Only 32-bit indices CUSPARSE_INDEX_32I is supported
- Only CSR format CUSPARSE_FORMAT_CSR is supported
- Only opA, opB equal to CUSPARSE_OPERATION_NON_TRANSPOSE are supported

The datatypes combinations currently supported for cusparseSpGEMM are listed below:

**Uniform-precision computation:**

<table>
<thead>
<tr>
<th>$A/B/C$/computeType</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_64F</td>
<td></td>
</tr>
</tbody>
</table>
cusparseSpGEMM routine runs for the following algorithm:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SPGEMM_DEFAULT</td>
<td>Default algorithm. Provides deterministic [bit-wise] results for each run</td>
</tr>
</tbody>
</table>

cusparseSpGEMM() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports does not support CUDA graph capture

cusparseSpGEMM() supports the following optimizations:

- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseSpGEMM for a code example.
Chapter 15. Appendix A: 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(), cudaMemcpy, and so on] that can be used to allocate memory on the GPU.

The cuSPARSE Fortran wrapper code is provided as an example only and needs to be compiled into an application for it to call the cuSPARSE API functions. Providing this source code allows users to make any changes necessary for a particular platform and toolchain.

The cuSPARSE Fortran wrapper code has been used to demonstrate interoperability with the compilers g95 0.91 (on 32-bit and 64-bit Linux) and g95 0.92 (on 32-bit and 64-bit Mac OS X). In order to use other compilers, users have to make any changes to the wrapper code that may be required.

The direct wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all cuSPARSE functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPU memory space [using CUDA_MALLOC() and CUDA_FREE()] and to copy data between GPU and CPU memory spaces [using the CUDA_MEMCPY() routines]. The sample wrappers provided in cusparse_fortran.c map device pointers to the OS-dependent type size_t, which is 32 bits wide on 32-bit platforms and 64 bits wide on a 64-bit platforms.

One approach to dealing with index arithmetic on device pointers in Fortran code is to use C-style macros and to use the C preprocessor to expand them. On Linux and Mac OS X, preprocessing can be done by using the option `-cpp` with g95 or gfortran. The function
GET_SHIFTED_ADDRESS(), provided with the cuSPARSE Fortran wrappers, can also be used, as shown in example B.

Example B shows the the C++ of example A implemented in Fortran 77 on the host. This example should be compiled with ARCH_64 defined as 1 on a 64-bit OS system and as undefined on a 32-bit OS system. For example, on g95 or gfortran, it can be done directly on the command line using the option -cpp -DARCH_64=1.

### 15.1. Fortran Application

```fortran
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|
     |    4.0        |
     |5.0     6.0 7.0|
     |    8.0     9.0|
c     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)
end do

c     create a sparse and dense vector
Appendix A: cuSPARSE Fortran Bindings

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) 

(none 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
xValHostPtr(2)=200.0
xValHostPtr(3)=400.0

print the vectors

do j=1,2
  do i=1,n
    write(*,*) "yHost Ptr[",i,"",",j,"]=",yHostPtr(i+n*(j-1))
  enddo
endo
do i=1,n
  write(*,*) "xIndHostPtr[i,],"=",xIndHostPtr(i)
  write(*,*) "xValHostPtr[i,]","=",xValHostPtr(i)
endo

allocate GPU memory and copy the matrix and vectors into it

cudaSuccess=0
cudaMempcpyHostToDevice=1
cudaStat1 = cuda_malloc(coorRowIndex,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 2
endif
cudaStat1 = cuda_mempcpy_fort2c_int(cooRowIndex,cooRowIndexHostPtr, nnz*4,1)
cudaStat2 = cuda_mempcpy_fort2c_int(cooColIndex,cooColIndexHostPtr, nnz*4,1)
cudaStat3 = cuda_mempcpy_fort2c_real(cooVal, cooValHostPtr, nnz*8,1)
cudaStat4 = cuda_mempcpy_fort2c_real(y, yHostPtr, 2*n*8,1)
cudaStat5 = cuda_mempcpy_fort2c_int(xInd, xIndHostPtr, nnz_vector*4,1)
cudaStat6 = cuda_mempcpy_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 1
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 1
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 1
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 1
endif
status = cusparse_set_mat_type(descrA,0)
status = cusparse_set_mat_index_base(descrA,1)
Appendix A: cuSPARSE Fortran Bindings

```
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 2
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 cusparse_destroy_mat_descr(descrA)
call cusparse_destroy(handle)
write(*,*) "Conversion from COO to CSR format failed"
stop 1
endif
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 cusparse_destroy_mat_descr(descrA)
call cusparse_destroy(handle)
write(*,*) "Scatter from sparse to dense vector failed"
stop 1
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, nnz, dtwo,
```

---

```
c                           ynp1, 1)
call cuda_free(cooRowIndex)
call cuda_free(cooColIndex)
call cuda_free(cooVal)
call cusparse_destroy_mat_descr(descrA)
call cusparse_destroy(handle)
write(*,*) "Scatter from sparse to dense vector failed"
stop 1
endif
```
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 1
endif

c     print intermediate results (y)
c     y = [10 20 30 40 | 680 760 1230 2240]
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 1
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 2
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 1
endif
status= cusparse_dcsrmv(handle, 0, n, 2, nnnz, 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-vector multiplication failed"
stop 1
endif
c     print final results (z)
c     cudaMemcpyDeviceToHost=0
c     cudaMemcpyDeviceToHost=2
     cudaStat1 = cudaMemcpy_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 1
endif
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 0
end
16.1. COO Sort

This chapter provides a simple example in the C programming language of sorting of COO format.

A is a 3x3 sparse matrix,

\[
A = \begin{pmatrix}
1.0 & 2.0 & 0.0 \\
0.0 & 5.0 & 0.0 \\
0.0 & 8.0 & 0.0 \\
\end{pmatrix}
\]

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    cudaError_t cudaStat6 = cudaSuccess;

    const int m = 3;

    /*
    * A is a 3x3 sparse matrix
    * | 1 2 0 |
    * | 0 5 0 |
    * | 0 8 0 |
    *
    * const int m = 3;
    */
```
const int n = 3;
const int nnz = 4;

#if 0
/* index starts at 0 */
int h_cooRows[nnz] = {2, 1, 0, 0};
int h_cooCols[nnz] = {1, 1, 0, 1};
#else
/* index starts at -2 */
int h_cooRows[nnz] = {0, -1, -2, -2};
int h_cooCols[nnz] = {-1, -1, -2, -1};
#endif

double h_cooVals[nnz] = {8.0, 5.0, 1.0, 2.0};
int h_P[nnz];

int *d_cooRows = NULL;
int *d_cooCols = NULL;
int *d_P = NULL;
double *d_cooVals = NULL;
double *d_cooVals_sorted = NULL;
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;

printf("m = %d, n = %d, nnz=%d \n", m, n, nnz);

/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);

status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: allocate buffer */
status = cusparseXcoosort_bufferSizeExt(
    handle,
    m,
    n,
    nnz,
    d_cooRows,
    d_cooCols,
    &pBufferSizeInBytes
);
assert( CUSPARSE_STATUS_SUCCESS == status);

printf("pBufferSizeInBytes = %lld bytes \n", (long long)pBufferSizeInBytes);

cudaStat1 = cudaMalloc( &d_cooRows, sizeof(int)*nnz);
cudaStat2 = cudaMalloc( &d_cooCols, sizeof(int)*nnz);
cudaStat3 = cudaMalloc( &d_P, sizeof(int)*nnz);
cudaStat4 = cudaMalloc( &d_cooVals, sizeof(double)*nnz);
cudaStat5 = cudaMalloc( &d_cooVals_sorted, sizeof(double)*nnz);
cudaStat6 = cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

assert( cudaSuccess == cudaStat1 );
assert( cudaSuccess == cudaStat2 );
assert( cudaSuccess == cudaStat3 );
assert( cudaSuccess == cudaStat4 );
assert( cudaSuccess == cudaStat5 );
assert( cudaSuccess == cudaStat6 );

cudaStat1 = cudaMemcpy(d_cooRows, h_cooRows, sizeof(int)*nnz ,
cudaMemcpyHostToDevice);

Appendix B: Examples of sorting

cudaStat2 = cudaMemcpy(d_cooCols, h_cooCols, sizeof(int)*nnz, cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_cooVals, h_cooVals, sizeof(double)*nnz, cudaMemcpyHostToDevice);
cudaStat4 = cudaDeviceSynchronize();
assert( cudaSuccess == cudaStat1 );
assert( cudaSuccess == cudaStat2 );
assert( cudaSuccess == cudaStat3 );
assert( cudaSuccess == cudaStat4 );

/* step 3: setup permutation vector P to identity */
status = cusparseCreateIdentityPermutation(
    handle,
    nnz,
    d_P);
assert( CUSPARSE_STATUS_SUCCESS == status);

/* step 4: sort COO format by Row */
status = cusparseXcoosortByRow(
    handle,
    m,
    n,
    nnz,
    d_cooRows,
    d_cooCols,
    d_P,
    pBuffer
);
assert( CUSPARSE_STATUS_SUCCESS == status);

/* step 5: gather sorted cooVals */
status = cusparseDgthr(
    handle,
    nnz,
    d_cooVals,
    d_cooVals_sorted,
    d_P,
    CUSPARSE_INDEX_BASE_ZERO
);
assert( CUSPARSE_STATUS_SUCCESS == status);

cudaStat1 = cudaDeviceSynchronize(); /* wait until the computation is done */
cudaStat2 = cudaMemcpy(h_cooRows, d_cooRows, sizeof(int)*nnz, cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(h_cooCols, d_cooCols, sizeof(int)*nnz, cudaMemcpyDeviceToHost);
cudaStat4 = cudaMemcpy(h_cooVals, d_cooVals_sorted, sizeof(double)*nnz, cudaMemcpyDeviceToHost);
cudaStat5 = cudaMemcpy(h_P, d_P, sizeof(int)*nnz, cudaMemcpyDeviceToHost);
cudaStat6 = cudaDeviceSynchronize();
assert( cudaSuccess == cudaStat1 );
assert( cudaSuccess == cudaStat2 );
assert( cudaSuccess == cudaStat3 );
assert( cudaSuccess == cudaStat4 );
assert( cudaSuccess == cudaStat5 );
assert( cudaSuccess == cudaStat6 );

printf("sorted coo: \n");
for(int j = 0 ; j < nnz; j++){
    printf("(%d, %d, %f) \n", h_cooRows[j], h_cooCols[j], h_cooVals[j] );
}

for(int j = 0 ; j < nnz; j++)
    printf("P[%d] = %d \n", j, h_P[j] );
Appendix B: Examples of sorting

```c
/* free resources */
if (d_cooRows)    cudaFree(d_cooRows);
if (d_cooCols)    cudaFree(d_cooCols);
if (d_P)          cudaFree(d_P);
if (d_cooVals)    cudaFree(d_cooVals);
if (d_cooVals_sorted) cudaFree(d_cooVals_sorted);
if (pBuffer)      cudaFree(pBuffer);
if (handle)       cusparseDestroy(handle);
if (stream)       cudaStreamDestroy(stream);
cudaDeviceReset();
return 0;
```
Chapter 17. Appendix C: Examples of prune

17.1. Prune Dense to Sparse

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix of CSR format.

A is a 4x4 dense matrix,

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            float Areg = A[row + col*lda];
            printf("%s%d,%d = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(int m, int n, int nnz, const cusparseMatDescr_t descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA,
```
Appendix C: Examples of prune

cuSPARSE Library

const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)? 0:1;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrC = NULL;
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int lda = m;
    /*
     *      | 1  0  2  -3 |
     *      | 0  4  0  0 |
     *  A = | 5  0  6  7 |
     *      | 0  8  0  9 |
     */
    const float A[lda*n] = {1, 0, 5, 0, 0, 4, 0, 8, 2, 0, 6, 0, -3, 0, 7, 9};
    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC = NULL;
    float *d_A = NULL;
    int *d_csrRowPtrC = NULL;
    int *d_csrColIndC = NULL;
    float *d_csrValC = NULL;
    size_t lworkInBytes = 0;
    char *d_work = NULL;
    int nnzC = 0;
    float threshold = 4.1; /* remove Aij <= 4.1 */
    // float threshold = 0; /* remove zeros */
    printf("example of pruneDense2csr \n");
    printf("prune |A(i,j)| <= threshold \n");
    printf("threshold = %f \n", threshold);
    printMatrix(m, n, A, lda, "A");
    /* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);

status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);

status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix C */
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);

cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);

cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(float)*lda*n);
cudaStat2 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

/* step 3: query workspace */
cudaStat1 = cudaMemcpy(d_A, A, sizeof(float)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

status = cusparseSpruneDense2csr_bufferSizeExt(handle,
m,
n,
d_A,
lda,
&threshold,
descrC,
d csrValC,
d csrRowPtrC,
d csrColIndC,
&wworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);

printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);

if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
status = cusparseSpruneDense2csrNnz(handle,
m,
n,
d_A,
lda,
&threshold,
descrC,
d csrRowPtrC,
&nnzC, /* host */
d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC){
Appendix C: Examples of prune

```c
printf("C is empty \n");
return 0;
}

/* step 5: compute csrColIndC and csrValC */
cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
cudaStat2 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
status = cusparseSpruneDense2csr(
    handle,
    m,
    n,
    d_A,
    lda,
    &threshold,
    descrC,
    d_csrValC,
    d_csrRowPtrC,
    d_csrColIndC,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);
/* step 6: output C */
csrRowPtrC = (int*  )malloc(sizeof(int  )*(m+1));
csrColIndC = (int*  )malloc(sizeof(int  )*nnzC);
csrValC    = (float*)malloc(sizeof(float)*nnzC);
assert( NULL != csrRowPtrC);
assert( NULL != csrColIndC);
assert( NULL != csrValC);
cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);
printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");
/* free resources */
if (d_A           ) cudaFree(d_A);
if (d_csrRowPtrC  ) cudaFree(d_csrRowPtrC);
if (d_csrColIndC  ) cudaFree(d_csrColIndC);
if (d_csrValC     ) cudaFree(d_csrValC);
if (csrRowPtrC    ) free(csrRowPtrC);
if (csrColIndC     ) free(csrColIndC);
if (csrValC       ) free(csrValC);
if (handle        ) cusparseDestroy(handle);
if (stream        ) cudaStreamDestroy(stream);
if (descrC        ) cusparseDestroyMatDescr(descrC);
cudaDeviceReset();
return 0;
```
17.2. Prune Sparse to Sparse

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix of CSR format.

A is a 4x4 sparse matrix,

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)? 0:1;
    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row] - base;
        const int end = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaMemcpy_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    return 0;
}
```

/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *  nvcc -c -I/usr/local/cuda/include prunecsr_example.cpp
 *  g++ -o prunecsr_example.cpp prunecsr_example.o -L/usr/local/cuda/lib64 -lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)? 0:1;
    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row] - base;
        const int end = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaMemcpy_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    return 0;
}
Appendix C: Examples of prune

cuSPARSE Library

const int nnzA = 9;
/*
 *      |    1     0     2     -3  |
 *      |    0     4     0     0   |
 *  A = |    5     0     6     7   |
 *      |    0     8     0     9   |
 */

const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

int* csrRowPtrC = NULL;
int* csrColIndC = NULL;
float* csrValC  = NULL;

int *d_csrRowPtrA = NULL;
int *d_csrColIndA = NULL;
float *d_csrValA = NULL;

int *d_csrRowPtrC = NULL;
int *d_csrColIndC = NULL;
float *d_csrValC = NULL;

size_t lworkInBytes = 0;
char *d_work = NULL;

int nnzC = 0;
float threshold = 4.1; /* remove Aij <= 4.1 */
/* remove zeros */
printf("example of pruneCsr2csr \n");
printf("prune |A(i,j)| <= threshold \n");
printf("threshold = %E \n", threshold);

/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A and C */
status = cusparseCreateMatDescr(&descrA);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);
printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");
cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValA, sizeof(float)*nnzA);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(m+1), cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrValA, csrValA, sizeof(float)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
status = cusparseSpruneCsr2csr_bufferSizeExt(
    handle,
    m,
    n,
    nnzA,
    descrA,
    d_csrValA,  
    d_csrRowPtrA,
    d_csrColIndA,
    &threshold,
    descrC,
    d_csrValC,  
    d_csrRowPtrC,
    d_csrColIndC,
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);

if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
status = cusparseSpruneCsr2csrNnz(
    handle,
    m,
    n,
    nnzA,
    descrA,
    d_csrValA,  
    d_csrRowPtrA,
    d_csrColIndA,
    &threshold,
    descrC,
    d_csrRowPtrC,
    &nnzC, /* host */
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC) {
    printf("C is empty \n");
    return 0;
}

/* step 5: compute csrColIndC and csrValC */
    cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int ) * nnzC);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC);
    assert(cudaSuccess == cudaStat1);

    status = cusparseSpruneCsr2csr(
        handle,
        m,
        n,
        nnzA,
        descrA,
        d_csrValA,
        d_csrRowPtrA,
        d_csrColIndA,
        &threshold,
        descrC,
        d_csrValC,
        d_csrRowPtrC,
        d_csrColIndC,
        d_work);
    assert(CUSPARSE_STATUS_SUCCESS == status);
    cudaStat1 = cudaDeviceSynchronize();
    assert(cudaSuccess == cudaStat1);

    /* step 6: output C */
    csrRowPtrC = (int*) malloc(sizeof(int )*(m+1));
    csrColIndC = (int*) malloc(sizeof(int )*nnzC);  
    csrValC = (float*) malloc(sizeof(float)*nnzC);
    assert( NULL != csrRowPtrC);
    assert( NULL != csrColIndC);
    assert( NULL != csrValC);
    cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int )*(m+1), cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int )*nnzC, cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    cudaStat1 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC , cudaMemcpyDeviceToHost);
    assert(cudaSuccess == cudaStat1);
    printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

    /* free resources */
    if (d_csrRowPtrA ) cudaFree(d_csrRowPtrA);
    if (d_csrColIndA) cudaFree(d_csrColIndA);
    if (d_csrValA) cudaFree(d_csrValA);
    if (d_csrRowPtrC ) cudaFree(d_csrRowPtrC);
    if (d_csrColIndC ) cudaFree(d_csrColIndC);
    if (d_csrValC) cudaFree(d_csrValC);
    if (csrRowPtrC ) free(csrRowPtrC);
    if (csrColIndC ) free(csrColIndC);
    if (csrValC ) free(csrValC);
    if (handle ) cusparseDestroy(handle);
    if (stream) cudaStreamDestroy(stream);
    if (descrA ) cusparseDestroyMatDescr(descrA);
    if (descrC ) cusparseDestroyMatDescr(descrC);
    cudaDeviceReset();
    return 0;
}
17.3. Prune Dense to Sparse by Percentage

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix by percentage.

A is a 4x4 dense matrix,

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

The percentage is 50, which means to prune 50 percent of the dense matrix. The matrix has 16 elements, so 8 out of 16 must be pruned out. Therefore 7 zeros are pruned out, and value 1.0 is also out because it is the smallest among 9 nonzero elements.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++) {
        for(int col = 0 ; col < n ; col++) {
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(int m, int n, int nnz, const cusparseMatDescr_t descrA, const float* csrValA, const int* csrRowPtrA, const int* csrColIndA, const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)? 0:1 ;
    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++) {
        const int start = csrRowPtrA[row] - base;
        const int end = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++) {
            printf("%s(%d,%d) = %f\n", name, row+1, colidx+1, csrValA[colidx]);
        }
    }
}
```
Appendix C: Examples of prune

```
const int col = csrColIndA[colidx] - base;
const float Areg = csrValA[colidx];
printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
}
}
}
}
}
int main(int argc, char*argv[])
{
  cusparseHandle_t handle = NULL;
cudaStream_t stream = NULL;
cusparseMatDescr_t descrC = NULL;
pruneInfo_t info = NULL;
cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;
cudaError_t cudaStat2 = cudaSuccess;
cudaError_t cudaStat3 = cudaSuccess;
cudaError_t cudaStat4 = cudaSuccess;
cudaError_t cudaStat5 = cudaSuccess;
const int m = 4;
const int n = 4;
const int lda = m;
/*
*      |    1     0     2     -3  |
*      |    0     4     0     0   |
*  A = |    5     0     6     7   |
*      |    0     8     0     9   |
*/
const float A[lda*n] = {1, 0, 5, 0, 0, 4, 0, 8, 2, 0, 6, 0, -3, 0, 7, 9};
int* csrRowPtrC = NULL;
int* csrColIndC = NULL;
float* csrValC = NULL;
float *d_A = NULL;
int *d_csrRowPtrC = NULL;
int *d_csrColIndC = NULL;
float *d_csrValC = NULL;
size_t lworkInBytes = 0;
char *d_work = NULL;
nonzC = 0;
float percentage = 50; /* 50% of nnz */
printf("example of pruneDense2csrByPercentage \n");
printf("prune out %.1f percentage of A \n", percentage);
printMatrix(m, n, A, lda, "A");
/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseCreatePruneInfo(&info);
assert(CUSPARSE_STATUS_SUCCESS == status);
```

Appendix C: Examples of prune

/* step 2: configuration of matrix C */
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);

cusparseSetMatIndexBase(descrC,CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL );

cudaStat1 = cudaMalloc ((void**)&d_A , sizeof(float)*lda*n );
cudaStat2 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

cudaStat1 = cudaMemcpy(d_A, A, sizeof(float)*lda*n, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
status = cusparseSpruneDense2csrByPercentage_bufferSizeExt(
    handle, 
    m, 
    n, 
    d_A, 
    lda, 
    percentage, 
    descrC, 
    d_csrValC, 
    d_csrRowPtrC, 
    d_csrColIndC, 
    info, 
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes = %lld \n", (long long)lworkInBytes);

if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
status = cusparseSpruneDense2csrNnzByPercentage{
    handle, 
    m, 
    n, 
    d_A, 
    lda, 
    percentage, 
    descrC, 
    d_csrRowPtrC, 
    &nnzC, /* host */
    info,  
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC ){
    printf("C is empty \n");
    return 0;
}

/* step 5: compute csrColIndC and csrValC */
cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int ) * nnzC );
cudaStat2 = cudaMalloc ((void**)&d_csrValC , sizeof(float) * nnzC );
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
status = cusparseSpruneDense2csrByPercentage(
    handle,
    m,
    n,
    d_A,
    lda,
    percentage,
    descrC,
    d_csrValC,
    d_csrRowPtrC,
    d_csrColIndC,
    info,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 7: output C */
csrRowPtrC = (int*)malloc(sizeof(int)*(m+1));
csrColIndC = (int*)malloc(sizeof(int)*nnzC);
csrValC = (float*)malloc(sizeof(float)*nnzC);
assert(NULL != csrRowPtrC);
assert(NULL != csrColIndC);
assert(NULL != csrValC);

cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int)*(m+1),
    cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int)*nnzC ,
    cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(csrValC , d_csrValC , sizeof(float)*nnzC ,
    cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
if (d_A) cudaFree(d_A);
if (d_csrRowPtrC) cudaFree(d_csrRowPtrC);
if (d_csrColIndC) cudaFree(d_csrColIndC);
if (d_csrValC) cudaFree(d_csrValC);
if (csrRowPtrC) free(csrRowPtrC);
if (csrColIndC) free(csrColIndC);
if (csrValC) free(csrValC);
if (handle) cusparseDestroy(handle);
if (stream) cudaStreamDestroy(stream);
if (descrC) cusparseDestroyMatDescr(descrC);
if (info) cusparseDestroyPruneInfo(info);

cudaDeviceReset();
return 0;
17.4. Prune Sparse to Sparse by Percentage

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix by percentage.

A is a 4x4 sparse matrix,

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

The percentage is 20, which means to prune 20 percent of the nonzeros. The sparse matrix has 9 nonzero elements, so 1.4 elements must be pruned out. The function removes 1.0 and 2.0 which are first two smallest numbers of nonzeros.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)?
    0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n,
    nnz, base);
    for(int row = 0 ; row < m ; row++)
    {
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++)
        {
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    return 0;
}
```

/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunecsr2csrByP.cpp
 *   g++ -o prunecsr2csrByP.cpp prunecsr2csrByP.o -L/usr/local/cuda/lib64 -lcusparse
 *   -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)?
    0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n,
    nnz, base);
    for(int row = 0 ; row < m ; row++)
    {
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++)
        {
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    return 0;
}
cudaStream_t stream = NULL;
cusparseMatDescr_t descrA = NULL;
cusparseMatDescr_t descrC = NULL;
pruneInfo_t info = NULL;

cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;
const int m = 4;
const int n = 4;
const int nnzA = 9;
/*
 *  | 1  0  2  -3 |
 *  | 0  4  0  0  |
 * A = | 5  0  6  7  |
 *  | 0  8  0  9  |
 */

const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

int* csrRowPtrC = NULL;
int* csrColIndC = NULL;
float* csrValC = NULL;
int *d_csrRowPtrA = NULL;
int *d_csrColIndA = NULL;
float *d_csrValA = NULL;
int *d_csrRowPtrC = NULL;
int *d_csrColIndC = NULL;
float *d_csrValC = NULL;
size_t lworkInBytes = 0;
char *d_work = NULL;
nnzC = 0;
float percentage = 20; /* remove 20% of nonzeros */
printf("example of pruneCsr2csrByPercentage \
");
printf("prune %.1f percent of nonzeros \n", percentage);

/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseCreatePruneInfo(&info);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix C */
status = cusparseCreateMatDescr(&descrA);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
Appendix C: Examples of prune

```
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);
printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");
```
Appendix C: Examples of prune

```
&nnzC, /* host */
info,
  d_work);

assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC ){
    printf("C is empty \n");
    return 0;
}

/* step 5: compute csrColIndC and csrValC */
cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int  ) * nnzC );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValC   , sizeof(float) * nnzC );
assert(cudaSuccess == cudaStat1);
status = cusparseSpruneCsr2csrByPercentage(
    handle, 
    m, 
    n, 
    nnzA, 
    descrA, 
    d_csrValA, 
    d_csrRowPtrA, 
    d_csrColIndA, 
    percentage, 
    descrC, 
    d_csrValC, 
    d_csrRowPtrC, 
    d_csrColIndC, 
    info, 
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 6: output C */
csrRowPtrC = (int*)malloc(sizeof(int  )*(m+1));
csrColIndC = (int*)malloc(sizeof(int  )*nnzC );
csrValC   = (float*)malloc(sizeof(float)*nnzC );
assert( NULL != csrRowPtrC);
assert( NULL != csrColIndC);
assert( NULL != csrValC);
cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(csrValC, d_csrValC, sizeof(float)*nnzC ,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
```

if (d_csrRowPtrA) cudaFree(d_csrRowPtrA);
if (d_csrColIndA) cudaFree(d_csrColIndA);
if (d_csrValA) cudaFree(d_csrValA);
if (d_csrRowPtrC) cudaFree(d_csrRowPtrC);
if (d_csrColIndC) cudaFree(d_csrColIndC);
if (d_csrValC) cudaFree(d_csrValC);

if (csrRowPtrC) free(csrRowPtrC);
if (csrColIndC) free(csrColIndC);
if (csrValC) free(csrValC);

if (handle) cusparseDestroy(handle);
if (stream) cudaStreamDestroy(stream);
if (descrA) cusparseDestroyMatDescr(descrA);
if (descrC) cusparseDestroyMatDescr(descrC);
if (info) cusparseDestroyPruneInfo(info);

cudaDeviceReset();

return 0;
Chapter 18. Appendix D: Examples of gpsv

18.1. Batched Penta-diagonal Solver

This section provides a simple example in the C programming language of gpsvInterleavedBatch.

The example solves two penta-diagonal systems and assumes data layout is NOT interleaved format. Before calling gpsvInterleavedBatch, cublasXgeam is used to transform the data layout, from aggregate format to interleaved format. If the user can prepare interleaved format, no need to transpose the data.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>
#include <cublas_v2.h>

/*
* compute | b - A*x|_inf
*/
void residaul_eval(
    int n,
    const float *ds,
    const float *dl,
    const float *d,
    const float *du,
    const float *dw,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    float r_nrminf = 0;
    for(int i = 0 ; i < n ; i++){
        float dot = 0;
        if (i > 1 ){
            dot += ds[i]*x[i-2];
        }
        r_nrminf = max(r_nrminf, |
```
if (i > 0 ){
    dot += dl[i]*x[i-1];
}
dot += d[i]*x[i];
if (i < (n-1 )){
    dot += du[i]*x[i+1];
}
if (i < (n-2 )){
    dot += dw[i]*x[i+2];
}
float ri = b[i] - dot;
r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
}
*r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
cusparseHandle_t cusparseH = NULL;
cublasHandle_t cublasH = NULL;
cudaStream_t stream = NULL;
cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;
const int n = 4;
const int batchSize = 2;
/*
 A1 = |  1   8   13  0  |       |  1 |       | -0.0592 |
    |  5   2   9  14 |, b1 = |  2 |, x1 = |  0.3428 |
    |  11  6   3  10 |       |  3 |       | -0.1295 |
    |  0  12   7  4 |       |  4 |       |  0.1982 |
*/
/*
 A2 = |  15  22  27  0  |       |  5 |       | -0.0012 |
    |  19  16  23  28 |, b2 = |  6 |, x2 = |  0.2792 |
    |  25  20  17  24 |       |  7 |       | -0.0416 |
    |  0  26  21  18 |       |  8 |       |  0.0898 |
*/
/*
 A = (ds, dl, d, du, dw), B and X are in aggregate format
*/
const float ds[n * batchSize] = { 0, 0, 11, 12, 0, 0, 25, 26};
const float dl[n * batchSize] = { 0, 5, 6, 7, 0, 19, 20, 21};
const float d[n * batchSize] = { 1, 2, 3, 4, 15, 16, 17, 18};
const float du[n * batchSize] = { 8, 9, 10, 0, 22, 23, 24, 0};
const float dw[n * batchSize] = {13,14, 0, 0, 27, 28, 0, 0};
const float B[n * batchSize] = { 1, 2, 3, 4, 5, 6, 7, 8};
float X[n * batchSize]; /* Xj = Aj \ Bj */
/* device memory
*(d_ds0, d_dl0, d_d0, d_du0, d_dw0) is aggregate format
*(d_ds, d_dl, d_d, d_du, d_dw) is interleaved format
*/
float *d_ds0 = NULL;
float *d_dl0 = NULL;
float *d_d0 = NULL;
float *d_du0 = NULL;
float *d_dw0 = NULL;
float *d_ds = NULL;
float *d_dl = NULL;
float *d_d = NULL;
float *d_du = NULL;
float *d_dw = NULL;
float *d_B = NULL;
float *d_X = NULL;
size_t lworkInBytes = 0;
char *d_work = NULL;
const float h_one = 1;
const float h_zero = 0;
int algo = 0; /* QR factorization */

printf("example of gpsv (interleaved format) \n");
printf("n = %d, batchSize = %d\n", n, batchSize);

/* step 1: create cusparse/cublas handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);

status = cusparseCreate(&cusparseH);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(cusparseH, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cublasCreate(&cublasH);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);
status = cublasSetStream(cublasH, stream);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* step 2: allocate device memory */
cudaStat1 = cudaMalloc ((void**)&d_ds0 , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dl0 , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_d0 , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_du0 , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dw0 , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_ds , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dl , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_d , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_du , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dw , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_B , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_X , sizeof(float)*n*batchSize);
assert(cudaSuccess == cudaStat1);

/* step 3: prepare data in device, interleaved format */
cudaStat1 = cudaMemcpy(d_ds0, ds, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_dl0, dl, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_d0 , d , sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_du0, du, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_dw0, dw, sizeof(float)*n*batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_B , B, sizeof(float)*n*batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize();
/* convert ds to interleaved format
 * ds = transpose(ds0) */
cublasStat = cublasSgeam(
cublasH,
CUBLAS_OP_T, /* transa */
CUBLAS_OP_T, /* transb, don't care */
batchSize, /* number of rows of ds */
&h_one,
d_ds0, /* ds0 is n-by-batchSize */
n, /* leading dimension of ds0 */
&h_zero,
NULL,


assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* convert dl to interleaved format
 * dl = transpose(dl0) */
cublasStat = cublasSgeam(
cublasH,
CUBLAS_OP_T, /* transa */
CUBLAS_OP_T, /* transb, don't care */
batchSize, /* number of rows of dl */
&h_one,
d_dl0, /* dl0 is n-by-batchSize */
n, /* leading dimension of dl0 */
&h_zero,
NULL,


assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* convert d to interleaved format
 * d = transpose(d0) */
cublasStat = cublasSgeam(
cublasH,
CUBLAS_OP_T, /* transa */
CUBLAS_OP_T, /* transb, don't care */
batchSize, /* number of rows of d */
&h_one,
d_d0, /* d0 is n-by-batchSize */
n, /* leading dimension of d0 */
&h_zero,
NULL,


assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* convert du to interleaved format */
* du = transpose(du0)
*/
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of du */
    n,         /* number of columns of du */
    &h_one,
    d_du0, /* du0 is n-by-batchSize */
    n, /* leading dimension of du0 */
    &h_zero,
    NULL,
    n,         /* don't care */
    d_du, /* du is batchSize-by-n */
    batchSize /* leading dimension of du */
);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* convert dw to interleaved format */
* dw = transpose(dw0)
*/
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of dw */
    n,         /* number of columns of dw */
    &h_one,
    d_dw0, /* dw0 is n-by-batchSize */
    n, /* leading dimension of dw0 */
    &h_zero,
    NULL,
    n,         /* don't care */
    d_dw, /* dw is batchSize-by-n */
    batchSize /* leading dimension of dw */
);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* convert B to interleaved format */
* X = transpose(B)
*/
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of X */
    n,         /* number of columns of X */
    &h_one,
    d_B, /* B is n-by-batchSize */
    n, /* leading dimension of B */
    &h_zero,
    NULL,
    n,         /* don't care */
    d_X, /* X is batchSize-by-n */
    batchSize /* leading dimension of X */
);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);

/* step 4: prepare workspace */
status = cusparseSgsvInterleavedBatch_bufferSizeExt(
cusparseH,
    algo,
    n,
Appendix D: Examples of gpsi

cuSPARSE Library

D.06709-001_v11.2   |   311

d_ds,
d_d1,
d_d,
d_du,
d_dw,
d_X,
batchSize,
lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes = %lld \n", (long long)lworkInBytes);
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 5: solve Aj*xj = bj */
status = cusparseSgpsvInterleavedBatch(
cusparseH,
algo,
n,
d_ds,
d_d1,
d_d,
d_du,
d_dw,
d_X,
batchSize,
d_work);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSPARSE_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);

/* step 6: convert X back to aggregate format */
/* B = transpose(X) */
cublasStat = cublasSgeam(
cublasH,
CUBLAS_OP_T, /* transa */
CUBLAS_OP_T, /* transb, don't care */
n, /* number of rows of B */
batchSize, /* number of columns of B */
&h_one,
d_X, /* X is batchSize-by-n */
batchSize, /* leading dimension of X */
&h_zero,
NULL, /* don't care */
d_B, /* B is n-by-batchSize */
n /* leading dimension of B */
);
assert(CUBLAS_STATUS_SUCCESS == cublasStat);
cudaDeviceSynchronize();

/* step 7: residual evaluation */
cudaStat1 = cudaMemcpy(X,  d_B, sizeof(float)*n*batchSize,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize();

printf("==== x1 = inv(A1)*b1 \n");
for(int j = 0 ; j < n; j++){
    printf("x1[%d] = %f \n", j, X[j]);
}
float r1_nrminf;
residual_eval("== x1 = inv(A1)*b1 \n");
for(int j = 0 ; j < n; j++){
    printf("x1[%d] = %f \n", j, X[j]);
}
n,
ds,
dl,
d,
du,
dw,
B,
X,
&rl_nrminf
);
printf("|b1 - A1*x1| = %E\n", rl_nrminf);

printf("\n==== x2 = inv(A2)*b2 \n");
for(int j = 0 ; j < n; j++){
    printf("x2[%d] = %f\n", j, X[n+j]);
}
float r2_nrminf;
residaul_eval(
    n,
    ds + n,
    dl + n,
    d + n,
    du + n,
    dw + n,
    B + n,
    X + n,
    &r2_nrminf
);
printf("|b2 - A2*x2| = %E\n", r2_nrminf);
/* free resources */
if (d_ds0       ) cudaFree(d_ds0);
if (d_dl0       ) cudaFree(d_dl0);
if (d_d0        ) cudaFree(d_d0);
if (d_du0       ) cudaFree(d_du0);
if (d_dw0       ) cudaFree(d_dw0);
if (d_ds        ) cudaFree(d_ds);
if (d_dl        ) cudaFree(d_dl);
if (d_d         ) cudaFree(d_d);
if (d_du        ) cudaFree(d_du);
if (d_dw        ) cudaFree(d_dw);
if (d_B         ) cudaFree(d_B);
if (d_X        ) cudaFree(d_X);
if (cusparseH   ) cusparseDestroy(cusparseH);
if (cublasH     ) cublasDestroy(cublasH);
if (stream      ) cudaStreamDestroy(stream);
cudaDeviceReset();
    return 0;
}
Chapter 19. Appendix E: Examples of csrsm2

19.1. Forward Triangular Solver

This section provides a simple example in the C programming language of csrsm2. The example solves a lower triangular system with 2 right hand side vectors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void residual_eval(
    int n,
    const cusparseMatDescr_t descrA,
    const float *csrVal,
    const int *csrRowPtr,
    const int *csrColInd,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)? 0:1;
    const int lower = (CUSPARSE_FILL_MODE_LOWER == cusparseGetMatFillMode(descrA))?
                      1:0;
    const int unit  = (CUSPARSE_DIAG_TYPE_UNIT == cusparseGetMatDiagType(descrA))?
                       1:0;

    float r_nrminf = 0;
    for(int row = 0; row < n; row++){
        const int start = csrRowPtr[row]   - base;
        const int end   = csrRowPtr[row+1] - base;
        float dot = 0;
        for(int colidx = start; colidx < end; colidx++){
            const int col = csrColInd[colidx] - base;
            float Aij = csrVal[colidx];
            float xj  = x[col];
```

if ( (row == col) && unit ) {
    Aij = 1.0;
}

int valid = (row >= col) && lower ||
            (row <= col) && !lower;
if ( valid ) {
    dot += Aij*xj;
}

float ri = b[row] - dot;
    r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
  cusparseHandle_t handle = NULL;
  cudaStream_t stream = NULL;
  cusparseMatDescr_t descrA = NULL;
  csrsm2Info_t info = NULL;
  cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
  cudaError_t cudaStat1 = cudaSuccess;
  const int nrhs = 2;
  const int n = 4;
  const int nnzA = 9;
  const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
  const float h_one = 1.0;
  
  /*
   *    | 1 0 2 -3 |
   *    | 0 4 0 0 |
   * A = | 5 0 6 7 |
   *    | 0 8 0 9 |
   */
  
  /* Regard A as a lower triangle matrix L with non-unit diagonal. */
  
  /* Given B = | 2 6 |, X = L \ B = | 0.5 1.5 |
   *    | 3 7 | -0.3333 -3 |
   *    | 4 8 | 0 -0.4444 |
   */
  
  const int csrRowPtrA[n+1] = { 1, 4, 5, 8, 10};
  const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
  const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};
  const float B[n*nrhs] = {1,2,3,4,5,6,7,8};
  float X[n*nrhs];

  int *d_csrRowPtrA = NULL;
  int *d_csrColIndA = NULL;
  float *d_csrValA = NULL;
  float *d_B = NULL;

  size_t lworkInBytes = 0;
  char *d_work = NULL;

  const int algo = 0; /* non-block version */

  printf("example of csrsm2 \n");

  /* step 1: create cusparse handle, bind a stream */
  cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
  assert(cudaSuccess == cudaStat1);

  status = cusparseCreate(&handle);
  assert(CUSPARSE_STATUS_SUCCESS == status);}
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

status = cusparseCreateCsrsm2Info(&info);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A */
/* A is base-1 */
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);

cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);

/* A is lower triangle */
cusparseSetMatFillMode(descrA, CUSPARSE_FILL_MODE_LOWER);

/* A has non unit diagonal */
cusparseSetMatDiagType(descrA, CUSPARSE_DIAG_TYPE_NON_UNIT);

cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(n+1) );
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMalloc ((void**)&d_csrValA , sizeof(float)*nnzA );
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMalloc ((void**)&d_B , sizeof(float)*n*nrhs );
assert(cudaSuccess == cudaStat1);

status = cusparseScsrsm2_bufferSizeExt(
    handle,
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n,
    nrhs,
    nnzA,
    &h_one,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    d_B,
    n,   /* ldb */
    info,
    policy,
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);

printf("lworkInBytes = %lld \n", (long long)lworkInBytes);
if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);

status = cusparseScsrsm2(handle, algo, algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n, nrhs, nnzA, &h_one, descrA,
    d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes,
    lworkInBytes, lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
    policy, &lworkInBytes, lworkInBytes, & lworkInBytes,
    & info, &h_one, descrA, d_csrValA, d_csrRowPtrA, d_csrColIndA, d_B, n,
assert(cudaSuccess == cudaStat1);

/* step 4: analysis */
status = cusparseScsrm2_analysis(
    handle,
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n,
    nrhs,
    nnzA,
    &h_one,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    d_B,
    n,   /* ldb */
    info,
    policy,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 5: solve L * X = B */
status = cusparseScsrm2_solve(
    handle,
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n,
    nrhs,
    nnzA,
    &h_one,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    d_B,
    n,   /* ldb */
    info,
    policy,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 6: measure residual B - A*X */
cudaStat1 = cudaMemcpy(X, d_B, sizeof(float)*n*nrhs, cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize();

printf("==== x1 = inv(A)*b1 \n");
for(int j = 0 ; j < n; j++){
    printf("x1[%d] = %f\n", j, X[j]);
}
float r1_nrminf;
residual_eval(
    n,
    descrA,
    csrValA,
    csrRowPtrA,
    csrColIndA,
    B,
    X,
    &r1_nrminf
);

// cuSPARSE Library
Appendix E: Examples of csrsm2

```c
printf("|b1 - A*x1| = %E\n", r1_nrminf);
printf("==== x2 = inv(A)*b2 \n");
for(int j = 0 ; j < n; j++){
    printf("x2[%d] = %f\n", j, X[n+j]);
}
float r2_nrminf;
residaul_eval(
    n,
    descrA,
    csrValA,
    csrRowPtrA,
    csrColIndA,
    B+n,
    X+n,
    &r2_nrminf
);
printf("|b2 - A*x2| = %E\n", r2_nrminf);

/* free resources */
if (d_csrRowPtrA  ) cudaFree(d_csrRowPtrA);
if (d_csrColIndA  ) cudaFree(d_csrColIndA);
if (d_csrValA     ) cudaFree(d_csrValA);
if (d_B           ) cudaFree(d_B);

if (handle        ) cusparseDestroy(handle);
if (stream        ) cudaStreamDestroy(stream);
if (descrA        ) cusparseDestroyMatDescr(descrA);
if (info          ) cusparseDestroyCsrsm2Info(info);

cudaDeviceReset();

return 0;
}```
Chapter 20. Appendix F: Acknowledgements

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

- The cusparse\texttt{gtsv} implementation is derived from a version developed by Li-Wen Chang from the University of Illinois.
- the cusparse\texttt{gtsvInterleavedBatch} adopts cuThomasBatch developed by Pedro Valero-Lara and Ivan Martínez-Pérez from Barcelona Supercomputing Center and BSC/UPC NVIDIA GPU Center of Excellence.
Chapter 21. Bibliography


Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

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

Copyright

© 2007-2020 NVIDIA Corporation. All rights reserved.