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
      3.2.2. 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 [DEPRECATED]..............................................................................................17
4.10. cusparseColorInfo_t..................................................................................................................17
4.11. cusparseSolvePolicy_t..............................................................................................................18
4.12. bsric02Info_t..........................................................................................................................18
4.13. bsrlu02Info_t..........................................................................................................................18
4.14. bsrsm2Info_t..........................................................................................................................18
4.15. bsrsv2Info_t...........................................................................................................................18
4.16. csrgemm2Info_t [DEPRECATED].............................................................................................18
4.17. csric02Info_t..........................................................................................................................19
4.18. csrlu02Info_t..........................................................................................................................19
4.19. csrsm2Info_t [DEPRECATED].................................................................................................19
4.20. csrsv2Info_t [DEPRECATED].................................................................................................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().......................................................................................................21
5.8. cusparseSetPointerMode()........................................................................................................21
5.9. cusparseGetStream()..............................................................................................................21
5.10. cusparseSetStream()...............................................................................................................21

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()...................................................................................................25
6.8. cusparseGetMatType()............................................................................................................25
6.9. cusparseSetMatDiagType().....................................................................................................25
6.10. cusparseSetMatFillMode()....................................................................................................25
6.11. cusparseSetMatIndexBase()................................................................................................25
6.12. cusparseSetMatType()...........................................................................................................25
6.13. cusparseCreateCsrsv2Info() [DEPRECATED]........................................................................28
6.14. cusparseDestroyCsrsv2Info() [DEPRECATED]......................................................................28
6.15. cusparseCreateCsrsm2Info() [DEPRECATED]........................................................................... 28
6.16. cusparseDestroyCsrsm2Info() [DEPRECATED]....................................................................... 29
6.17. cusparseCreateCsrco2Info()........................................................................................................ 29
6.18. cusparseDestroyCsrco2Info()...................................................................................................... 29
6.19. cusparseCreateCsrilu02Info()..................................................................................................... 30
6.20. cusparseDestroyCsrilu02Info().................................................................................................... 30
6.21. cusparseCreateBsrsv2Info()........................................................................................................ 30
6.22. cusparseDestroyBsrsv2Info()...................................................................................................... 31
6.23. cusparseCreateBsrsm2Info()......................................................................................................... 31
6.24. cusparseDestroyBsrsm2Info().................................................................................................... 31
6.25. cusparseCreateBsric02Info()..................................................................................................... 32
6.26. cusparseDestroyBsric02Info().................................................................................................. 32
6.27. cusparseCreateBsrilu02Info()................................................................................................... 33
6.28. cusparseDestroyBsrilu02Info()................................................................................................. 33
6.29. cusparseCreateCsrgemm2Info() [DEPRECATED]...................................................................... 33
6.30. cusparseDestroyCsrgemm2Info() [DEPRECATED].................................................................... 33
6.31. cusparseCreatePruneInfo()........................................................................................................ 33
6.32. cusparseDestroyPruneInfo()..................................................................................................... 34

Chapter 7. cuSPARSE Level 1 Function Reference...........................................................................35
7.1. cusparse<t>axpyi() [DEPRECATED]............................................................................................ 35
7.2. cusparse<t>gthr() [DEPRECATED]............................................................................................... 36
7.3. cusparse<t>gthrz() [DEPRECATED].............................................................................................. 37
7.4. cusparse<t>roti() [DEPRECATED]................................................................................................. 39
7.5. cusparse<t>sctr() [DEPRECATED].................................................................................................. 40

Chapter 8. cuSPARSE Level 2 Function Reference...........................................................................42
8.1. cusparse<t>bsrmv()....................................................................................................................... 42
8.2. cusparse<t>bsrxmv().................................................................................................................... 45
8.3. cusparse<t>bsrv2_bufferSize()..................................................................................................... 49
8.4. cusparse<t>bsrv2_analysis()........................................................................................................ 51
8.5. cusparse<t>bsrv2_solve()............................................................................................................. 54
8.6. cusparseXbsrv2_zeroPivot()......................................................................................................... 58
8.7. cusparse<ct>csrsv2_bufferSize() [DEPRECATED]...................................................................... 59
8.8. cusparse<t>csrsv2_analysis() [DEPRECATED]........................................................................... 61
8.9. cusparse<t>csrsv2_solve() [DEPRECATED].................................................................................. 65
8.10. cusparse<t>csrsv2_zeroPivot() [DEPRECATED]......................................................................... 69
8.11. cusparse<ct>gemvi().................................................................................................................. 70

Chapter 9. cuSPARSE Level 3 Function Reference..........................................................................73
14.1.2. cusparseFormat_t ......................................................................................................... 238
14.1.3. cusparseOrder_t ........................................................................................................ 238
14.1.4. cusparseIndexType_t ............................................................................................... 239
14.2. Sparse Vector APIs ...................................................................................................... 239
   14.2.1. cusparseCreateSpVec() ........................................................................................... 239
   14.2.2. cusparseDestroySpVec() .......................................................................................... 239
   14.2.3. cusparseSpVecGet() .................................................................................................. 240
   14.2.4. cusparseSpVecGetIndexBase() .................................................................................. 240
   14.2.5. cusparseSpVecGetValues() ...................................................................................... 241
   14.2.6. cusparseSpVecSetValues() ...................................................................................... 241
14.3. Sparse Matrix APIs ...................................................................................................... 241
   14.3.1. cusparseCreateCoo() ............................................................................................... 241
   14.3.2. cusparseCreateCooAoS() [DEPRECATED] .................................................................. 242
   14.3.3. cusparseCreateCsr() ............................................................................................... 243
   14.3.4. cusparseCreateCsc() ............................................................................................... 243
   14.3.5. cusparseCreateBlockedEll() .................................................................................... 244
   14.3.6. cusparseDestroySpMat() .......................................................................................... 245
   14.3.7. cusparseCooGet() ..................................................................................................... 246
   14.3.8. cusparseCooAosGet() [DEPRECATED] ....................................................................... 246
   14.3.9. cusparseCsrGet() ..................................................................................................... 247
   14.3.10. cusparseCsrSetPointers() ...................................................................................... 248
   14.3.11. cusparseCscSetPointers() ...................................................................................... 248
   14.3.12. cusparseCooSetPointers() ...................................................................................... 248
   14.3.13. cusparseBlockedEllGet() ....................................................................................... 249
   14.3.14. cusparseSpMatGetSize() .......................................................................................... 250
   14.3.15. cusparseSpMatGetFormat() .................................................................................... 250
   14.3.16. cusparseSpMatGetIndexBase() .............................................................................. 250
   14.3.17. cusparseSpMatGetValues() .................................................................................... 251
   14.3.18. cusparseSpMatSetValues() .................................................................................... 251
   14.3.19. cusparseSpMatGetStridedBatch() ........................................................................... 251
   14.3.20. cusparseSpMatSetStridedBatch() [DEPRECATED] .................................................. 252
   14.3.21. cusparseCooSetStridedBatch() ............................................................................... 252
   14.3.22. cusparseCsrSetStridedBatch() ............................................................................... 252
   14.3.23. cusparseSpMatGetAttribute() ................................................................................. 253
   14.3.24. cusparseSpMatSetAttribute() ................................................................................. 253
14.4. Dense Vector APIs ....................................................................................................... 254
   14.4.1. cusparseCreateDnVec() ........................................................................................... 254
   14.4.2. cusparseDestroyDnVec() .......................................................................................... 254
14.4.3. cusparseDnVecGet()......................................................................................................255
14.4.4. cusparseDnVecGetValues()........................................................................................255
14.4.5. cusparseDnVecSetValues()........................................................................................255

14.5. Dense Matrix APIs...............................................................................................................256
14.5.1. cusparseCreateDnMat()................................................................................................256
14.5.2. cusparseDestroyDnMat().............................................................................................. 256
14.5.3. cusparseDnMatGet()..................................................................................................... 257
14.5.4. cusparseDnMatGetValues().......................................................................................... 257
14.5.5. cusparseDnSetValues().................................................................................................257
14.5.6. cusparseDnMatGetStridedBatch()............................................................................... 258
14.5.7. cusparseDnMatSetStridedBatch()............................................................................. 258

14.6. Generic API Functions ........................................................................................................ 258
14.6.1. cusparseSparseToDense()............................................................................................259
14.6.2. cusparseDenseToSparse()............................................................................................260
14.6.3. cusparseAxpby()............................................................................................................ 262
14.6.4. cusparseGather().......................................................................................................... 263
14.6.5. cusparseScatter().......................................................................................................... 264
14.6.6. cusparseRot()................................................................................................................ 265
14.6.7. cusparseSpVV()............................................................................................................. 267
14.6.8. cusparseSpMV()............................................................................................................ 269
14.6.9. cusparseSpSV()............................................................................................................. 272
14.6.10. cusparseSpMM()......................................................................................................... 275
14.6.11. cusparseSpSM().......................................................................................................... 280
14.6.12. cusparseConstrainedGeMM() [DEPRECATED].......................................................... 283
14.6.13. cusparseSDDMM()...................................................................................................... 284
14.6.14. cusparseSpGEMM().....................................................................................................286
14.6.15. cusparseSpGEMMreuse()........................................................................................... 289

Chapter 15. Appendix A: cuSPARSE Fortran Bindings ..................................................................293
15.1. Fortran Application.............................................................................................................294

Chapter 16. Appendix B: Examples of sorting ...........................................................................302
16.1. COO Sort..............................................................................................................................302

Chapter 17. Appendix C: Examples of prune ............................................................................306
17.1. Prune Dense to Sparse.........................................................................................................306
17.2. Prune Sparse to Sparse.........................................................................................................310
17.3. Prune Dense to Sparse by Percentage............................................................................ 314
17.4. Prune Sparse to Sparse by Percentage...........................................................................318

Chapter 18. Appendix D: Examples of gpsv .............................................................................323
18.1. Batched Penta-diagonal Solver..................................................................................323

Chapter 19. Appendix E: Examples of csrsm2.................................................................330
  19.1. Forward Triangular Solver.....................................................................................330

Chapter 20. Appendix F: Acknowledgements................................................................335

Chapter 21. Bibliography.................................................................................................336
List of Figures

Figure 1. Blocked-ELL representation .......................................................................................... 245
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 cudaMemcpy(), 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<t>[<matrix data format>]<operation>[<output matrix data format>]

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

The <matrix data format> can be dense, coo, csr, 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:

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

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

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

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

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

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

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

2.1. Thread Safety

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

2.2. Scalar Parameters

In the cuSPARSE API, the scalar parameters \( \alpha \) 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
Using the cuSPARSE API

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.  

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:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 2.0 & 3.0 & 0.0 & 4.0
\end{bmatrix}
\]

(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.

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0
\end{bmatrix}
\begin{bmatrix}
1 & 4 & 5 & 7
\end{bmatrix}
\]
It can also be stored as a sparse vector with zero-based indexing.

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

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

### 3.2. 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.

| \( m \) | (integer) | The number of rows in the matrix. |
| \( n \) | (integer) | The number of columns in the matrix. |
| \( 1dX \) | (integer) | The leading dimension of \( X \), which must be greater than or equal to \( m \). If \( 1dX \) is greater than \( m \), then \( X \) represents a sub-matrix of a larger matrix stored in memory. |
| \( X \) | (pointer) | Points to the data array containing the matrix elements. It is assumed that enough storage is allocated for \( X \) to hold all of the matrix elements and that cuSPARSE library functions may access values outside of the sub-matrix, but will never overwrite them. |

For example, \( m \times n \) dense matrix \( X \) with leading dimension \( 1dX \) can be stored with one-based indexing as shown.

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

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

\[
\begin{bmatrix}
X_{1,1} & X_{2,1} & \ldots & X_{m,1} & X_{1dX,1} & \ldots & X_{1,n} & X_{2,n} & \ldots & X_{m,n} & X_{1dX,n}
\end{bmatrix}
\]

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

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

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

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

Consider again the $4 \times 5$ 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.

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

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

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

### 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>nnz</th>
<th>[integer]</th>
<th>The number of nonzero elements in the matrix.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cscValA</td>
<td>[pointer]</td>
<td>Points to the data array of length nnz that holds all nonzero values of $A$ in column-major format.</td>
</tr>
<tr>
<td>cscRowIndA</td>
<td>[pointer]</td>
<td>Points to the integer array of length nnz that contains the row indices of the corresponding elements in array cscValA.</td>
</tr>
<tr>
<td>cscColPtrA</td>
<td>[pointer]</td>
<td>Points to the integer array of length $n+1$ that holds indices into the arrays cscRowIndA and cscValA. The first $n$ entries of this array contain the indices of the first nonzero element in the $i$th row for $i=1,\ldots,n$, 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.

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

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

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

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>Parameter</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 ( \text{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 ( \text{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 ( \text{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 $2\times3$ block matrix $A_b$. The dimension of $A_b$ is $4\times6$, 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.

```plaintext
bsrValA = [A_{00}, A_{01}, A_{10}, A_{11}, A_{12}]
bsrRowPtrA = [0, 2, 5]
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 `bsrValA` is this.

```plaintext
bsrValA=[1 0 4 2 0 3 0 0 5 0 0 0 0 9 7 0 8 6 0 0 ]
```

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

```plaintext
bsrValA=[1 4 0 2 0 0 3 0 5 0 0 0 0 7 9 0 8 0 6 0 ]
```

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

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

```plaintext
bsrValA=[1 0 4 2 0 3 0 0 5 0 0 0 0 9 7 0 8 6 0 0 ]
```

```plaintext
bsrRowPtrA = [1 3 6]
bsrColIndA = [1 2 1 2 3]
```

**Note:** The general BSR format has two parameters, `rowBlockDim` and `colBlockDim`. `rowBlockDim` is number of rows within a block and `colBlockDim` is number of columns within a block. If `rowBlockDim=colBlockDim`, general BSR format is the same as BSR format. If
cuSPARSE Indexing and Data Formats

rowBlockDim=colBlockDim=1, general BSR format is the same as CSR format. The conversion routine gebsr2gebsr is used to do conversion among CSR, BSR and general BSR.

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

3.2.6. 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 \( \mathbf{A} \) is represented in BSRX format by the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockDim</td>
<td>(integer) Block dimension of matrix ( \mathbf{A} ).</td>
</tr>
<tr>
<td>mb</td>
<td>(integer) The number of block rows of ( \mathbf{A} ).</td>
</tr>
<tr>
<td>nb</td>
<td>(integer) The number of block columns of ( \mathbf{A} ).</td>
</tr>
<tr>
<td>nnzb</td>
<td>(integer) number of nonzero blocks in the matrix ( \mathbf{A} ).</td>
</tr>
<tr>
<td>bsrValA</td>
<td>(pointer) Points to the data array of length ( \text{nnzb} \times \text{blockDim}^2 ) that holds all the elements of the nonzero blocks of ( \mathbf{A} ). The block elements are stored in either column-major order or row-major order.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>(pointer) Points to the integer array of length ( \text{mb} ) that holds indices into the arrays bsrColIndA and bsrValA; bsrRowPtrA(( i )) is the position of the first nonzero block of the ( i )th block row in bsrColIndA and bsrValA.</td>
</tr>
<tr>
<td>bsrEndPtrA</td>
<td>(pointer) Points to the integer array of length ( \text{mb} ) that holds indices into the arrays bsrColIndA and bsrValA; bsrRowPtrA(( i )) is the position next to the last nonzero block of the ( i )th block row in bsrColIndA and bsrValA.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>(pointer) Points to the integer array of length ( \text{nnzb} ) that contains the column indices of the corresponding blocks in array bsrValA.</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 \( \mathbf{A}_b \) represented as shown.

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

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.

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

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 \( \hat{A} \).

\[
\begin{align*}
\text{bsrValA of } \hat{A} &= [A_{00} \quad A_{01} \quad A_{10} \quad A_{11} \quad A_{12}] \\
\text{bsrColIndA of } \hat{A} &= [0 \quad 1 \quad 0 \quad 1 \quad 2] \\
\text{bsrRowPtrA of } \hat{A} &= [0 \quad 3] \\
\text{bsrEndPtrA of } \hat{A} &= [0 \quad 4]
\end{align*}
\]
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>CUSPARSE_STATUS_NOT_INITIALIZED</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>To correct: call cusparseCreate() prior to the function call; and check that the hardware, an appropriate version of the driver, and the cuSPARSE library are correctly installed.</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_ALLOC_FAILED</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>To correct: prior to the function call, deallocate previously allocated memory as much as possible.</td>
</tr>
<tr>
<td>CUSPARSE_STATUS_INVALID_VALUE</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[SID]C|Z]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>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CUSPARSE_DIRECTION_COLUMN</td>
<td>the matrix should be parsed by columns.</td>
</tr>
</tbody>
</table>

### 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 $y = Ax$ when $A$ is symmetric and only lower triangular part is stored, two steps are needed. First step is to compute $y = (L+D) \times x$ and second step is to compute $y = L^T \times x + y$. Given the fact that the transpose operation $y = L^T \times x$ is 10x slower than non-transpose version $y = L \times x$, 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 $y = Ax$ 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 [DEPRECATED]

[[DEPRECATED]] The routine will be removed in the next major release

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`, `bsrv2`, `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. **bsrv2Info_t**

This is a pointer type to an opaque structure holding the information used in `bsrv2_bufferSize()`, `bsrv2_analysis()`, and `bsrv2_solve()`.

4.16. **csrgemm2Info_t [DEPRECATED]**

[[DEPRECATED]] *The routine will be removed in the next major release*

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 [DEPRECATED]
[[DEPRECATED]] The routine will be removed in the next major release
This is a pointer type to an opaque structure holding the information used in csrsm2_bufferSize(), csrsm2_analysis(), and csrsm2_solve().

4.20. csrsv2Info_t [DEPRECATED]
[[DEPRECATED]] The routine will be removed in the next major release
This is a pointer type to an opaque structure holding the information used in csrsv2_bufferSize(), csrsv2_analysis(), and csrsv2_solve().
Chapter 5.  cuSPARSE Management
Function Reference

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

5.1.  cusparseCreate()

cusparseStatus_t
cusparseCreate(cusparseHandle_t *handle)

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

<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
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()

cusparseStatus_t
cusparseGetErrorName(const char* errorName)

This function returns the string description of the cuSPARSE error status. It checks the error status in the specified handle and returns the corresponding string description.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorName</td>
<td>IN</td>
<td>The error status string</td>
</tr>
</tbody>
</table>
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()

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()

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()**

```c
 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()**

```c
 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()**

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

This function returns the version number of the cuSPARSE library.

<table>
<thead>
<tr>
<th>Param.</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>IN</td>
<td>cuSPARSE handle</td>
</tr>
<tr>
<td>version</td>
<td>OUT</td>
<td>The version number of the library</td>
</tr>
</tbody>
</table>

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

```
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()

cusparseStatus_t
cusparseCreateColorInfo(cusparseColorInfo_t* info)

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

Input

| info          | the pointer to the cusparseColorInfo_t structure |

See cusparseStatus_t for the description of the return status.

6.2. cusparseCreateMatDescr()

cusparseStatus_t
cusparseCreateMatDescr(cusparseMatDescr_t *descrA)

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

Input

| descrA              | the pointer to the matrix descriptor. |

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()`

```c
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()`

```c
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()`

```c
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()**

```c
CUSPARSE_INDEXBASE_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()**

```c
CUSPARSE_MATRIX_TYPE_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()**

```c
CUSPARSE_STATUS_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()**

```c
CUSPARSE_STATUS_T
cusparseSetMatFillMode(cusparseMatDescr_t descrA,
                       cusparseFillMode_t fillMode)
```

...
6.10. `cusparseSetFillMode()`

```
cusparseFillMode_t fillMode)
```

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillMode</td>
<td>One of the enumerated fillMode types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>descrA</td>
<td>the matrix descriptor.</td>
</tr>
</tbody>
</table>

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

6.11. `cusparseSetMatIndexBase()`

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

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

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>One of the enumerated indexBase types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>descrA</td>
<td>the matrix descriptor.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>One of the enumerated matrix types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>descrA</td>
<td>the matrix descriptor.</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
6.13. **cusparseCreateCsrsv2Info()**  [DEPRECATED]

**[[DEPRECATED]]  The routine will be removed in the next major release**

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

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

**Input**

<table>
<thead>
<tr>
<th>info</th>
<th>the pointer to the solve and analysis structure of csrsv2.</th>
</tr>
</thead>
</table>

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

6.14. **cusparseDestroyCsrsv2Info()**  [DEPRECATED]

**[[DEPRECATED]]  The routine will be removed in the next major release**

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

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

**Input**

<table>
<thead>
<tr>
<th>info</th>
<th>the solve (csrsv2_solve) and analysis (csrsv2_analysis) structure.</th>
</tr>
</thead>
</table>

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

6.15. **cusparseCreateCsrsm2Info()**  [DEPRECATED]

**[[DEPRECATED]]  The routine will be removed in the next major release**

```c
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() [DEPRECATED]

[[DEPRECATED]] The routine will be removed in the next major release

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. |
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()**

cusparseStatus_t

cusparseDestroyCsrilu02Info(csrilu02Info_t info);

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

**Input**

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

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

6.21. **cusparseCreateBsrsv2Info()**

cusparseStatus_t

cusparseCreateBsrsv2Info(bsrsv2Info_t *info);

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

**Input**

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

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

6.22. **cusparseDestroyBsrsv2Info()**

cusparseStatus_t

...
cusparseDestroyBsrsv2Info(bsrsv2Info_t info);

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

**Input**

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

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

### 6.23. cusparseCreateBsrsm2Info()

cusparseStatus_t cusparseCreateBsrsm2Info(bsrsm2Info_t *info);

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

**Input**

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

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

### 6.24. cusparseDestroyBsrsm2Info()

cusparseStatus_t cusparseDestroyBsrsm2Info(bsrsm2Info_t info);

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

**Input**

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

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

### 6.25. cusparseCreateBsric02Info()

cusparseStatus_t cusparseCreateBsric02Info(bsric02Info_t *info);

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

**Input**

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

```c
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()**  
**[DEPRECATED]**

The routine will be removed in the next major release.

```c
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()**  
**[DEPRECATED]**

The routine will be removed in the next major release.

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

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

**Input**

| info | opaque structure of csrgemm2. |

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

6.31. **cusparseCreatePruneInfo()**

```c
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 cusparseAxpy() instead. The routine will be removed in the next major release

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

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

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

```c
cusparseStatus_t
cusparseZaxpyi(cusparseHandle_t       handle,
              int                    nnz,
              const cuDoubleComplex* alpha,
              const cuDoubleComplex* xVal,
              const cuDoubleComplex* 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

$$y = y + \alpha x$$

In other words,

$$y[xInd[i]-idxBase] = y[xInd[i]-idxBase] + \alpha xVal[i]$$

- 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>nnz</td>
<td>number of elements in vector $x$.</td>
</tr>
<tr>
<td>alpha</td>
<td>$&lt;$type$&gt;$ scalar used for multiplication.</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$&gt;$ 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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>$&lt;$type$&gt;$ updated vector in dense format (that is unchanged if $nnz == 0$).</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

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

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

This function gathers the elements of the vector \( y \) listed in the index array \( x\text{Ind} \) 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>Parameter</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})-\text{idxBase}+1 )].</td>
</tr>
<tr>
<td>xInd</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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xVal</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 \( \text{cusparseStatus\_t} \) for the description of the return status.

7.3. \( \text{cusparse\textless t\textgreater gthrz()} \) [DEPRECATED]

[DEPRECATED] use \( \text{cusparseGather()} \) 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 $x\text{Ind}$ into the data array $x\text{Val}$. 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>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;type&gt; vector in dense format [of size$\geq$max($x\text{Ind}$)-idxBase+1].</td>
</tr>
<tr>
<td>xInd</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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xVal</td>
<td>&lt;type&gt; vector with nnz nonzero values that were gathered from vector $y$ [that is unchanged if nnz == 0].</td>
</tr>
</tbody>
</table>
<type> vector in dense format with elements indexed by xInd set to zero (it is unchanged if nnz == 0).

See cusparseStatus_t for the description of the return status.

7.4. cusparse<t>roti() [DEPRECATED]

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

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,

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

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

Input

<table>
<thead>
<tr>
<th>Parameter</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; 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)
 cusparseStatus_t cusparseDsctr(cusparseHandle_t handle, int nnz, const double* xVal, const int* xInd, double* y, cusparseIndexBase_t idxBase)
 cusparseStatus_t cusparseCsctr(cusparseHandle_t handle, int nnz, const cuComplex* xVal, const int* xInd, cuComplex* y, cusparseIndexBase_t idxBase)
 cusparseStatus_t cusparseZsctr(cusparseHandle_t handle, int nnz, const cuDoubleComplex* xVal, const int* xInd, cuDoubleComplex* y, cusparseIndexBase_t idxBase)
```

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

- 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>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>Output</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.
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 \texttt{csrsv2\_analysis()} function. The analysis is specific to the sparsity pattern of the given matrix and to the selected \texttt{cusparseOperation\_t} type. The information from the analysis phase is stored in the parameter of type \texttt{csrsv2Info\_t} that has been initialized previously with a call to \texttt{cusparseCreateCsrsv2Info()}.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the \texttt{csrsv2Info\_t} parameter by calling the appropriate \texttt{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 \texttt{csrsv2Info\_t} parameter can be released by calling \texttt{cusparseDestroyCsrsv2Info()}.

### 8.1. \texttt{cusparse\textless t\textgreater bsrmv()} 

\begin{verbatim}
cusparse\textless t\textgreater bsrmv(cusparseHandle_t handle, 
cusparseDirection_t dir, 
cusparseOperation_t trans, 
int mb, 
int nb, 
nznb, 
const float* alpha, 
const cusparseMatDescr_t descr, 
const float* bsrVal, 
const int* bsrRowPtr, 
const int* bsrColInd, 
int blockDim, 
const float* x, 
const float* beta,

cusparseStatus_t

cusparse\textless t\textgreater bsrmv(cusparseHandle_t handle, 
cusparseDirection_t dir, 
cusparseOperation_t trans, 
int mb, 
int nb, 
nznb, 
const float* alpha, 
const cusparseMatDescr_t descr, 
const float* bsrVal, 
const int* bsrRowPtr, 
const int* bsrColInd, 
int blockDim, 
const float* x, 
const float* beta,
\end{verbatim}
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 \((nb \cdot \text{blockDim})\) at least, and the size of vector y should be \((mb \cdot \text{blockDim})\) 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));
cusparseScsr2bsrNnz(handle, dirA, m, n,
                    descrA, csrRowPtrA, csrColIndA, blockDim,
                    descrC, bsrRowPtrC, &nnzb);
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dirA, m, n,
                 descrA, csrValA, csrRowPtrA, csrColIndA, blockDim,
                 descrC, bsrValC, bsrRowPtrC, bsrColIndC);
// step 2: allocate vector x and vector y large enough for bsrmv

cudaMalloc((void**)&x, sizeof(float)*(nb*blockDim));
cudaMalloc((void**)&y, sizeof(float)*(mb*blockDim));
cudaMemcpy(x, hx, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(y, hy, sizeof(float)*m, cudaMemcpyHostToDevice);
// step 3: perform bsrmv

cusparseScsr2bsr(handle, dirA, transA, mb, nb, nnzb, &alpha,
                  descrC, bsrValC, bsrRowPtrC, bsrColIndC, blockDim, x, &beta, y);
```

Input

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
</tbody>
</table>
trans | the operation \( \text{op}(A) \). Only \nCUSPARSE_OPERATION_NON_TRANSPOSE is \nsupported.
---|---
mb | number of block rows of matrix \( A \).
nb | number of block columns of matrix \( A \).
nnzb | number of nonzero blocks of matrix \( A \).
alpha | \(<\text{type}>\) scalar used for multiplication.
descr | the descriptor of matrix \( A \). The supported matrix \ntype is \nCUSPARSE_MATRIX_TYPE_GENERAL. \nAlso, the supported index bases are \nCUSPARSE_INDEX_BASE_ZERO \nand \nCUSPARSE_INDEX_BASE_ONE.
bsrVal | \(<\text{type}>\) array of \( \text{nnz} = \text{csrRowPtrA(mb)} - \n\text{csrRowPtrA(0)} \) nonzero blocks of matrix \( A \).
bsrRowPtr | integer array of \( \text{mb} + 1 \) elements that contains the \nstart of every block row and the end of the last \nblock row plus one.
bsrColInd | integer array of \( \text{nnz} = \text{csrRowPtrA(mb)} - \n\text{csrRowPtrA(0)} \) column indices of the nonzero \nblocks of matrix \( A \).
blockDim | block dimension of sparse matrix \( A \), larger than \nzero.
x | \(<\text{type}>\) vector of \( \text{nb} \times \text{blockDim} \) elements.
beta | \(<\text{type}>\) scalar used for multiplication. If \( \beta \) is \nzero, \( y \) does not have to be a valid input.
y | \(<\text{type}>\) vector of \( \text{mb} \times \text{blockDim} \) elements.

Output

| y | \(<\text{type}>\) updated vector.

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

### 8.2. \texttt{cusparse\_t\_bsrXmv()}
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,
const int* bsrColInd,
int blockDim,
const double* x,
const double* beta,
double* y)

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

cusparseStatus_t
cusparseZbsrxmv(cusparseHandle_t handle,
cusparseDirection_t dir,
cusparseOperation_t trans,
int sizeOfMask,
int mb,
int nb,
int nnzb,
const cuDoubleComplex* alpha,
const cusparseMatDescr_t descr,
const cuDoubleComplex* bsrVal,
const int* bsrMaskPtr,
const int* bsrRowPtr,
const int* bsrEndPtr,
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 `bsrVal`, `bsrRowPtr`, `bsrEndPtr`, and `bsrColInd`; \( x \) and \( y \) are vectors; \( \alpha \) and \( \beta \) are scalars; 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 mask operation is defined by array `bsrMaskPtr` which contains updated block row indices of \( y \). If row \( i \) is not specified in `bsrMaskPtr`, then `bsrmv()` does not touch row block \( i \) of \( A \) and \( y \).

For example, consider the 2x3 block matrix \( A \):

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

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

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

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

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

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

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

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

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

The mask operator is equivalent to the following operation:
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.

\[
\begin{bmatrix}
? \\
y_2
\end{bmatrix} := alpha \times \begin{bmatrix}
? \\
O \\
A_{22} \\
O \\
? \\
\end{bmatrix} \times \begin{bmatrix}
X_1 \\
x_2 \\
x_3
\end{bmatrix} + beta \times \begin{bmatrix}
? \\
y_2
\end{bmatrix}
\]

`bsrXmv()` 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 `bsrXmv()`:

- 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>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>dir</td>
<td>storage format of blocks, either <code>CUSPARSE_DIRECTION_ROW</code> or <code>CUSPARSE_DIRECTION_COLUMN</code>.</td>
</tr>
<tr>
<td>trans</td>
<td>the operation <code>op(A)</code>. Only <code>CUSPARSE_OPERATION_NON_TRANSPOSE</code> is supported.</td>
</tr>
<tr>
<td>sizeOfMask</td>
<td>number of updated block rows of <code>y</code>.</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of matrix <code>A</code>.</td>
</tr>
<tr>
<td>nb</td>
<td>number of block columns of matrix <code>A</code>.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of nonzero blocks of matrix <code>A</code>.</td>
</tr>
<tr>
<td>alpha</td>
<td><code>&lt;type&gt;</code> scalar used for multiplication.</td>
</tr>
<tr>
<td>descr</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>bsrVal</td>
<td><code>&lt;type&gt;</code> array of 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 the every block row plus one.

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

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

x: <type> vector of nb * blockDim elements.

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

y: <type> vector of mb * blockDim elements.

See cusparseStatus_t for the description of the return status.

8.3. cusparse<t>bsrsv2_bufferSize()

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

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

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

\( A \) 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`; \( x \) 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 `trans` and the upper [lower] triangular part of \( A \), `bsrsv2_bufferSize()` returns the maximum size buffer among these combinations. The buffer size depends on the dimensions `mb`, `blockDim`, and the number of nonzero blocks of the matrix `nnzb`. If the user changes the matrix, it is necessary to call `bsrsv2_bufferSize()` again to have the correct buffer size; otherwise a segmentation fault may occur.

- 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>dirA</code></td>
<td>storage format of blocks, either CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN.</td>
</tr>
<tr>
<td><code>transA</code></td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td><code>mb</code></td>
<td>number of block rows of matrix ( A ).</td>
</tr>
<tr>
<td><code>nnzb</code></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**

**info**

record of internal states based on different algorithms.

**pBufferSizeInBytes**

number of bytes of the buffer used in the bsrsv2_analysis() and bsrsv2_solve().

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

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

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

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

$A$ 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 $bsrColIndA$; $x$ 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 of BSR format is of size $blockDim \times blockDim$, 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.

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

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

Function $bsrsv2\_analysis()$ 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)

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

 cusparseStatus_t cusparseCbsrsv2_solve(cusparseHandle_t handle,
                                        cusparseDirection_t dirA,
                                        cusparseOperation_t transA,
                                        int mb,
                                        int nnzb,
                                        const cuComplex* alpha,
                                        const cusparseMatDescr_t descrA,
                                        const cuComplex* bsrValA,
                                        const int* bsrRowPtrA,
                                        const int* bsrColIndA,
```

This function performs the solve phase of `bsrsv2`, a new sparse triangular linear system
\[ \text{op}(A) \cdot y = \alpha \cdot x. \]

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

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

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 `CUSPARSE_DIRECTION_COLUMN` or `CUSPARSE_DIRECTION_ROW`. The matrix type must be `CUSPARSE_MATRIX_TYPE_GENERAL`, and the fill mode and diagonal type are ignored. Function `bsrsv02_solve()` can support an arbitrary \( \text{blockDim} \).

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

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

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

The level information may not improve the performance, but may spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism.
In this case, \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL} performs better than \texttt{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL}. If the user has an iterative solver, the best approach is to do \texttt{bsrsv2\_analysis()} with \texttt{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL} once. Then do \texttt{bsrsv2\_solve()} with \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL} in the first run, and with \texttt{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL} in the second run, and pick the fastest one to perform the remaining iterations.

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

Function \texttt{bsrsv2\_solve()} reports the first numerical zero, including a structural zero. No numerical zero is reported if \texttt{CUSPARSE\_DIAG\_TYPE\_UNIT} is specified, even if \(A(j,j)\) is not invertible for some \(j\). The user needs to call \texttt{cusparseXbsrsv2\_zeroPivot()} to know where the numerical zero is.

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

- 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 \(L\cdot y = x\) by level information.

```c
// Suppose that \(L\) is \(m \times m\) 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 \texttt{cusparseCreate()},
// - (\texttt{d\_bsrRowPtr}, \texttt{d\_bsrColInd}, \texttt{d\_bsrVal}) is BSR of \(L\) on device memory,
// - \(\texttt{d\_x}\) is right hand side vector on device memory.
// - \(\texttt{d\_y}\) is solution vector on device memory.
// - \(\texttt{d\_x}\) and \(\texttt{d\_y}\) are of size \(m\).
// \texttt{cusparseMatDescr\_t descr = 0;}
// \texttt{bsrsv2\_Info\_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 \texttt{CUSPARSE\_DIAG\_TYPE\_UNIT}
// - (\(L\) may not have all diagonal elements.)
// \texttt{cusparseCreateMatDescr(&descr)};
// \texttt{cusparseSetMatIndexBase(descr, CUSPARSE\_INDEX\_BASE\_ONE)};
// \texttt{cusparseSetMatFillMode(descr, CUSPARSE\_FILL\_MODE\_LOWER)};
// \texttt{cusparseSetMatDiagType(descr, CUSPARSE\_DIAG\_TYPE\_UNIT)};

// step 2: create a empty info structure
```
cuSPARSE Level 2 Function Reference

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>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirA</td>
<td>storage format of blocks, either</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_DIRECTION_ROW or</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>matrix type is CUSPARSE_MATRIX_TYPE_GENERAL,</td>
</tr>
<tr>
<td></td>
<td>while the supported diagonal types</td>
</tr>
<tr>
<td></td>
<td>are CUSPARSE_DIAG_TYPE_UNIT and</td>
</tr>
<tr>
<td></td>
<td>CUSPARSE_DIAG_TYPE_NON_UNIT.</td>
</tr>
<tr>
<td>bsrValA</td>
<td>&lt;type&gt; array of nnzb (= bsrRowPtrA(mb) -</td>
</tr>
<tr>
<td></td>
<td>bsrRowPtrA(0)) nonzero blocks of matrix A.</td>
</tr>
<tr>
<td>bsrRowPtrA</td>
<td>integer array of mb + 1 elements that contains the</td>
</tr>
<tr>
<td></td>
<td>start of every block row and the end of the last</td>
</tr>
<tr>
<td></td>
<td>block row plus one.</td>
</tr>
<tr>
<td>bsrColIndA</td>
<td>integer array of nnzb (= bsrRowPtrA(mb) -</td>
</tr>
<tr>
<td></td>
<td>bsrRowPtrA(0)) column indices of the nonzero</td>
</tr>
<tr>
<td></td>
<td>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

```c
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 cudaMemcpy() 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() [DEPRECATED]

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

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

```c
 cusparseStatus_t cusparseCsrmvEx(cusparseHandle_t handle,
                     cusparseAlgMode_t alg,
                     cusparseOperation_t transA,
                     int m,
                     int n,
                     int nnz,
                     const void* alpha,
                     cudaDataType alphatype,
                     const cusparseMatDescr_t descrA,
                     const void* csrValA,
                     cudaDataType csrValAtype,
                     const int* csrRowPtrA,
                     const int* csrColIndA,
                     const void* x,
                     cudaDataType xtype,
                     const void* beta,
                     cudaDataType betatype,
                     void* y,
                     cudaDataType ytype,
                     cudaDataType executiontype,
                     void* buffer)
```
This function performs the matrix-vector operation

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

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 \( \text{cusparseCsrmvEx_bufferSize} \) returns the size of the workspace needed by \( \text{cusparseCsrmvEx} \).

The function has the following limitations:

- All pointers should be aligned with 128 bytes
- Only \( \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \) operation is supported
- Only \( \text{CUSPARSE\_MATRIX\_TYPE\_GENERAL} \) matrix type is supported
- Only \( \text{CUSPARSE\_INDEX\_BASE\_ZERO} \) 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 \( \text{cusparseCsrmvEx} \)

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

See \( \text{cusparseStatus\_t} \) for the description of the return status
8.8.  cusparse<t>csrsv2_bufferSize()  
[DEPRECATED]

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

This function returns the size of the buffer used in csrsv2, a new sparse triangular linear system \( op(A) \cdot y = \alpha \cdot x \).
A is an \( m \times m \) sparse matrix that is defined in CSR storage format by the three arrays \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \); \( x \) and \( y \) are the right-hand-side and the solution vectors; \( \alpha \) is a scalar; and

\[
\text{op}(A) = \begin{cases} 
  A & \text{if trans} = \text{CUSPARSE OPERATION NON TRANSPOSE} \\
  A^T & \text{if trans} = \text{CUSPARSE OPERATION TRANSPOSE} \\
  A^H & \text{if trans} = \text{CUSPARSE OPERATION CONJUGATE TRANSPOSE}
\end{cases}
\]

Although there are six combinations in terms of the parameter \( \text{trans} \) and the upper (lower) triangular part of \( A \), \( \text{csrsv2_bufferSize()} \) 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 \( \text{csrsv2_bufferSize()} \) 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>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>( \text{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>( \text{nnz} )</td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td>( \text{descrA} )</td>
<td>the descriptor of matrix ( A ). The supported matrix type is ( \text{CUSPARSE MATRIX TYPE GENERAL} ), while the supported diagonal types are ( \text{CUSPARSE DIAG TYPE UNIT} ) and ( \text{CUSPARSE DIAG TYPE NON_UNIT} ).</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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>record of internal states based on different algorithms.</td>
</tr>
<tr>
<td>( \text{pBufferSizeInBytes} )</td>
<td>number of bytes of the buffer used in the ( \text{csrsv2_analysis} ) and ( \text{csrsv2_solve} ).</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus_t} for the description of the return status.
8.9. cusparse<t>csrsv2_analysis()

[DEPRECATED]

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

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

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

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

cusparseStatus_t cusparseZcsrsv2_analysis(cusparseHandle_t handle,
cusparseOperation_t transA,
int m,
int nnz,
const cusparseMatDescr_t descrA,
const cuDoubleComplex* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
csrsv2Info_t info,
cusparseSolvePolicy_t policy,
This function performs the analysis 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 \( \text{csrValA} \), \( \text{csrRowPtrA} \), and \( \text{csrColIndA} \); \( x \) 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`.

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>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 <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;type&gt; 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 \( \text{nnz}(\equiv \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) \) column indices of the nonzero elements of matrix A.

### info
structure initialized using \( \text{cusparseCreateCsrsv2Info()} \).

### policy
The supported policies are \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \) and \( \text{CUSPARSE_SOLVE_POLICY_USE_LEVEL} \).

### pBuffer
buffer allocated by the user, the size is returned by \( \text{csrsv2_bufferSize()} \).

### 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

### 8.10. cusparse<t>csrsv2_solve()

**[DEPRECATED]**

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

```c

#define csrsv2_solve_t cusparseStatus_t
cusparseStatus_t
cusparseScsrsv2_solve(cusparseHandle_t handle,
cusparseOperation_t transA,
int m,
int nnz,
const float* alpha,
const cusparseMatDescr_t descra,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
csrsv2Info_t info,
const float* x,
float* y,
cusparseSolvePolicy_t policy,
void* pBuffer)

cusparseStatus_t
cusparseDcsrsv2_solve(cusparseHandle_t handle,
cusparseOperation_t transA,
int m,
int nnz,
const double* alpha,
const cusparseMatDescr_t descra,
const double* csrValA,
```
This function performs the solve phase of \( \text{csrsv2} \), a new sparse triangular linear system \( \op(A) \cdot y = \alpha \cdot x \).

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

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

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

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

Although \( \text{csrsv2_solve()} \) can be done without level information, the user still needs to be aware of consistency. If \( \text{csrsv2_analysis()} \) is called with policy...
CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL, \texttt{csrsv2\_solve()} can be run with or without levels. On the contrary, if \texttt{csrsv2\_analysis()} is called with CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL, \texttt{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 \texttt{csrsv2\_analysis()} with CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL once. Then do \texttt{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 \texttt{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 \texttt{cusparseXcsrsv2\_zeroPivot()} to know where the numerical zero is.

For example, suppose \(L\) is a lower triangular matrix with unit diagonal, the following code solves \(L^\ast y = x\) 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;
csrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL;
cusparseOperation_t trans = CUSPARSE\_OPERATION\_NON\_TRANSPOSE;

// step 1: create a descriptor which contains 
// - matrix L is base-1 
// - matrix L is lower triangular 
// - matrix L has unit diagonal, specified by parameter CUSPARSE\_DIAG\_TYPE\_UNIT 
// (L may not have all diagonal elements.)
cusparseCreateMatDescr(&descr);
cusparseSetMatIndexBase(descr, CUSPARSE\_INDEX\_BASE\_ONE);
cusparseSetMatFillMode(descr, CUSPARSE\_FILL\_MODE\_LOWER);
cusparseSetMatDiagType(descr, CUSPARSE\_DIAG\_TYPE\_UNIT);

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

// step 3: query how much memory used in csrsv2, and allocate the buffer 
cusparseDcsrsv2\_bufferSize(handle, trans, m, nnz, descr, 
    d_csrVal, d_csrRowPtr, d_csrColInd, 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
", 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
", 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 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;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>csrValA</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>
</tbody>
</table>
### Info

<table>
<thead>
<tr>
<th>info</th>
<th>structure with information collected during the analysis phase [that should have been passed to the solve phase unchanged].</th>
</tr>
</thead>
</table>

### X

<table>
<thead>
<tr>
<th>x</th>
<th><code>&lt;type&gt;</code> right-hand-side vector of size $m$.</th>
</tr>
</thead>
</table>

### Policy

<table>
<thead>
<tr>
<th>policy</th>
<th>The supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.</th>
</tr>
</thead>
</table>

### pBuffer

<table>
<thead>
<tr>
<th>pBuffer</th>
<th>buffer allocated by the user, the size is return by csrsv2_bufferSize.</th>
</tr>
</thead>
</table>

### Output

<table>
<thead>
<tr>
<th>y</th>
<th><code>&lt;type&gt;</code> solution vector of size $m$.</th>
</tr>
</thead>
</table>

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

## 8.11. `cusparseXcsrsv2_zeroPivot()` [DEPRECATED]

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

```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

<table>
<thead>
<tr>
<th>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>info</th>
<th>info contains structural zero or numerical zero if the user already called csrsv2_analysis() or csrsv2_solve().</th>
</tr>
</thead>
</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 \texttt{cusparseStatus_t} for the description of the return status.

### 8.12. \texttt{cusparse<t>gemvi()}

```c
 cusparseStatus_t cusparseSgemvi_bufferSize(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, int* pBufferSize);
 cusparseStatus_t cusparseDgemvi_bufferSize(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, int* pBufferSize);
 cusparseStatus_t cusparseCgemvi_bufferSize(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, int* pBufferSize);
 cusparseStatus_t cusparseZgemvi_bufferSize(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, 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
```

---

\texttt{cusparse<t>gemvi()}
This function performs the matrix-vector operation

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

\( A \) 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 \( nnz \), and \( y \) is a dense vector; \( \alpha \) and \( \beta \) are scalars; and

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

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 \texttt{csr2csc()} functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.

2. Call the \texttt{gemvi()} function with the \texttt{cusparseOperation_t} parameter set to \texttt{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 \texttt{cusparse\_t\_gemvi\_bufferSize()} returns size of buffer used in \texttt{cusparse\_t\_gemvi()}

**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>trans</td>
<td>the operation $\texttt{op}(\texttt{A})$.</td>
</tr>
<tr>
<td>m</td>
<td>number of rows of matrix $\texttt{A}$.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of matrix $\texttt{A}$.</td>
</tr>
<tr>
<td>alpha</td>
<td>\texttt{&lt;type&gt;} scalar used for multiplication.</td>
</tr>
<tr>
<td>A</td>
<td>the pointer to dense matrix $\texttt{A}$.</td>
</tr>
<tr>
<td>lda</td>
<td>size of the leading dimension of $\texttt{A}$.</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzero elements of vector $\texttt{x}$.</td>
</tr>
<tr>
<td>x</td>
<td>\texttt{&lt;type&gt;} sparse vector of $\texttt{nnz}$ elements of size $\texttt{m}$ if $\texttt{op}(\texttt{A}) = \texttt{A}$, and size $\texttt{m}$ if $\texttt{op}(\texttt{A}) = \texttt{A}^T$ or $\texttt{op}(\texttt{A}) = \texttt{A}^H$.</td>
</tr>
<tr>
<td>xInd</td>
<td>Indices of non-zero values in $\texttt{x}$.</td>
</tr>
<tr>
<td>beta</td>
<td>\texttt{&lt;type&gt;} scalar used for multiplication. If $\texttt{beta}$ is zero, $\texttt{y}$ does not have to be a valid input.</td>
</tr>
<tr>
<td>y</td>
<td>\texttt{&lt;type&gt;} dense vector of $\texttt{m}$ elements if $\texttt{op}(\texttt{A}) = \texttt{A}$, and $\texttt{m}$ elements if $\texttt{op}(\texttt{A}) = \texttt{A}^T$ or $\texttt{op}(\texttt{A}) = \texttt{A}^H$.</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 \texttt{cusparse_t_gemvi()}.</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>\texttt{&lt;type&gt;} updated dense vector.</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus\_t} for the description of the return status.
Chapter 9. cuSPARSE Level 3 Function Reference

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

In particular, the solution of sparse triangular linear systems with multiple right-hand sides is implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate `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()`

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,
const int*               bsrColIndA,
int                      blockDim,
const cuDoubleComplex*   B,
int                      ldb,
const cuDoubleComplex*   beta,
cuDoubleComplex*         C,
int                      ldc)

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

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

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

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

and

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

The function has the following limitations:

- Only CUSPARSE_MATRIX_TYPE_GENERAL matrix type is supported
- Only blockDim > 1 is supported

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

In practice, no operation in an iterative solver or eigenvalue solver uses \( A \cdot \text{transpose}(B) \). However, we can perform \( A \cdot \text{transpose}(\text{transpose}(B)) \) which is the same as \( A \cdot B \). 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 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 \( A \cdot B \), our proposal is to transpose \( B \) to \( B^\top \) by first calling cublas<t>geam(), and then to perform \( A \cdot \text{transpose}(B^\top) \).
// 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

<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>transA</td>
<td>the operation op(A).</td>
</tr>
<tr>
<td>transB</td>
<td>the operation op(B).</td>
</tr>
<tr>
<td>mb</td>
<td>number of block rows of sparse matrix A.</td>
</tr>
<tr>
<td>n</td>
<td>number of columns of dense matrix op(B) and A.</td>
</tr>
<tr>
<td>kb</td>
<td>number of block columns of sparse matrix A.</td>
</tr>
<tr>
<td>nnzb</td>
<td>number of non-zero blocks of sparse 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. 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 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>
</tbody>
</table>
### cuSPARSE Level 3 Function Reference

#### bsrColIndA
- Integer array of `nnzb` columns indices of the non-zero blocks of matrix `A`.

#### blockDim
- Block dimension of sparse matrix `A`, larger than zero.

#### B
- Array of dimensions `(ldb, n)` if `op(B)=B` and `(ldb, k)` otherwise.

#### ldb
- Leading dimension of `B`. If `op(B)=B`, it must be at least `max(1, k)` if `op(B) != B`, it must be at least `max(1, n)`.

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

#### C
- Array of dimensions `(ldc, n)`.

#### ldc
- Leading dimension of `C`. It must be at least `max(1, m)` if `op(A)=A` and at least `max(1, k)` otherwise.

### Output

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

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,
int nnzb,
const cusparseMatDescr_t descrA,
const 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,
int nnzb,
const cusparseMatDescr_t descrA,
double* bsrSortedValA,
bsrSortedRowPtrA,
```
This function returns size of buffer used in `bsrsm2()`, a new sparse triangular linear system
\[ \text{op}(A) \ast \text{op}(X) = \text{op}(B) \].

`A` is an \((mb \ast \text{blockDim}) \times (mb \ast \text{blockDim})\) 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

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

Although there are six combinations in terms of parameter `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>Variable</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} (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 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, 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} (\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

<table>
<thead>
<tr>
<th>Variable</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 bsrsm2_analysis() and bsrsm2_solve().</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status

### 9.3. cusparse<t>bsrsm2_analysis()

cusparse<t>bsrsm2_analysis (handle, dirA, transA, transX, mb, n, nnzb, descrA, bsrSortedVal)
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 \((mb*\text{blockDim}) \times (mb*\text{blockDim})\) sparse matrix that is defined in BSR storage format by the three arrays \(\text{bsrValA}, \text{bsrRowPtrA},\) and \(\text{bsrColIndA}\); \(B\) and \(X\) are the right-hand-side and the solution matrices; \(\alpha\) is a scalar; and

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

and

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

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 CUSPARSE_DIRECTION_ROW or CUSPARSE_DIRECTION_COLUMN. The matrix type must be CUSPARSE_MATRIX_TYPE_GENERAL, and the fill mode and diagonal type are ignored.

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

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

Function \(\text{bsrsm2_analysis()}\) reports a structural zero and computes the level information stored in opaque structure \(\text{info}\). 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 CUSPARSE_SOLVE_POLICY_NO_LEVEL.

Function \(\text{bsrsm2_analysis()}\) always reports the first structural zero, even if the parameter \(\text{policy}\) is CUSPARSE_SOLVE_POLICY_NO_LEVEL. Besides, no structural zero is reported if CUSPARSE_DIAG_TYPE_UNIT is specified, even if block \(A(j,j)\) is missing for some \(j\). The user must call \(\text{cusparseXbsrsm2_query_zero_pivot()}\) 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>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>
</tbody>
</table>
n
number of columns of matrix op (B) and op (X).

nnzb
number of non-zero blocks of matrix A.

descrA
the descriptor of matrix A. The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL, while the supported diagonal types are CUSPARSE_DIAG_TYPE_UNIT and CUSPARSE_DIAG_TYPE_NON_UNIT.

bsrValA
<type> array of \( \text{nnzb} = \text{bsrRowPtrA}(\text{mb}) - \text{bsrRowPtrA}(0) \) 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 initialized using cusparseCreateBsrsm2Info.

policy
The supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.

pBuffer
buffer allocated by the user; the size is return by bsrsm2_bufferSize().

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

See cusparseStatus_t for the description of the return status.

9.4. cusparse<t>bsrsm2_solve()
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 float* B, int ldb, float* 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 double* B, int ldb, double* 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:

$$\text{op}(A) \ast \text{op}(X) = \alpha \ast \text{op}(B)$$

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

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

and

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

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 $\text{blockDim} \ast \text{blockDim}$, stored as column-major or row-major as determined by parameter $\text{dirA}$, 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 $\text{bsrsm2_solve()}$ 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 the buffer size returned by $\text{bsrsm2_bufferSize()}$. The address of $\text{pBuffer}$ must be multiple of 128 bytes. If it is not, CUSPARSE_STATUS_INVALID_VALUE is returned.

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

Function $\text{bsrsm2_solve()}$ has the same behavior as $\text{bsrsv02_solve()}$, reporting the first numerical zero, including a structural zero. The user must call $\text{cusparseXbsrsm2_query_zero_pivot()}$ to know where the numerical zero is.
The motivation of \texttt{transpose(X)} is to improve the memory access of matrix \( X \). The computational pattern of \texttt{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 \( p\text{Buffer} \neq \text{NULL} \):

\begin{itemize}
  \item The routine requires no extra storage
  \item The routine supports asynchronous execution
  \item The routine supports CUDA graph capture
\end{itemize}

\textbf{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>transX</td>
<td>the operation ( op(B) ) and ( 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>nnzb</td>
<td>number of non-zero blocks 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)) ) 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 ( 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 \texttt{cusparseCreateBsrsm2Info().}</td>
</tr>
<tr>
<td>B</td>
<td>(&lt;type&gt; right-hand-side array.)</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of ( B ). If ( op(B)=B ), ( ldb \geq (mb*blockDim) ); otherwise, ( ldb \geq n ).</td>
</tr>
<tr>
<td>ldx</td>
<td>leading dimension of ( X ). If ( op(X)=X ), then ( ldx \geq (mb*blockDim) ). Otherwise ( ldx \geq n ).</td>
</tr>
</tbody>
</table>
### 9.5. `cusparseXbsrsm2_zeroPivot()`

```c
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 <code>bsrsm2_analysis()</code> or <code>bsrsm2_solve()</code>.</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. `cusparse<t>csrsm2_bufferSizeExt()`  

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

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

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

```c
 cusparseStatus_t cusparseCcsrsm2_bufferSizeExt(cusparseHandle_t handle, 
 int algo, 
 cusparseOperation_t transA, 
 cusparseOperation_t transB, 
 int m, 
 int nrhs, 
 int nnz, 
 const cuComplex* alpha, 
 const cusparseMatDescr_t descrA, 
 const cuComplex* csrSortedValA, 
 const int* csrSortedRowPtrA, 
```

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

\( A \) 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

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

- The 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>algo = 0 is non-block version; algo = 1 is block version.</td>
</tr>
<tr>
<td><code>transA</code></td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td><code>transB</code></td>
<td>the operation ( \text{op}(B) ).</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td><code>nrhs</code></td>
<td>number of columns of right hand side matrix ( \text{op}(B) ).</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>(&lt;\text{type}&gt;) 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

A `<type>` array of $\text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ 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 $\text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))$ column indices of the nonzero elements of matrix $A$.

$B$

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

ldb

The 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

The 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()

[DEPRECATED]

Use [cusparseSpSM()](#) instead. The routine will be removed in the next major release.

cusparseStatus_t cusparseCcsrsm2_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,
cuSPARSE Library

**cuSPARSE Level 3 Function Reference**

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

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

cusparseStatus_t
cusparseZcsrsm2_analysis(cusparseHandle_t handle,
int algo,
cusparseOperation_t transA,
cusparseOperation_t transB,
int m,
int nrhs,
int nnz,
const cuDoubleComplex* alpha,
const cusparseMatDescr_t descrA,
const cuDoubleComplex* descrA,
const cuDoubleComplex* csrSortedValA,
const int* csrSortedRowPtrA,
const int* csrSortedColIndA,
const cuDoubleComplex* B,
int ldb,
This function performs the analysis 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

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

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

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

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

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

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

**Input**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>csrsm2Info_t info</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>cusparseSolvePolicy_t policy</code></td>
<td>algo = 0 is non-block version; algo = 1 is block version.</td>
</tr>
<tr>
<td><code>void* pBuffer</code></td>
<td>the operation <code>\text{op}(A)</code>.</td>
</tr>
<tr>
<td><code>transA</code></td>
<td>the operation <code>\text{op}(B)</code>.</td>
</tr>
<tr>
<td><code>transB</code></td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of columns of matrix ( \text{op}(B) ).</td>
</tr>
<tr>
<td><code>nrhs</code></td>
<td>number of nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td><code>nnz</code></td>
<td>&lt;type&gt; 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>, while the supported diagonal types are <code>CUSPARSE_DIAG_TYPE_UNIT</code> and <code>CUSPARSE_DIAG_TYPE_NON_UNIT</code>.</td>
</tr>
</tbody>
</table>
### csrValA

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

### csrRowPtrA

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

### csrColIndA

Integer array of `nnz(= csrRowPtrA(m) - csrRowPtrA(0))` column indices of the nonzero elements of matrix `A`.

### B

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

### ldb

Leading dimension of `B` and `X`.

### info

Structure initialized using `cusparseCreateCsrsv2Info()`.

### policy

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

### pBuffer

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

### Output

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<t>csrsm2_solve()`

**[DEPRECATED]**

Use `cusparseSpSM()` instead. The routine will be removed in the next major release.

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

---

**cuSPARSE Library**

DU-06709-001_v11.4 | 92
This function performs the solve phase of \( \text{csrsm2} \), a sparse triangular linear system \( \text{\( \text{opp}(A) \) \( \text{opp}(X) = \alpha \text{opp}(B) \)} \).
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

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

$\text{transB}$ acts on both matrix $B$ and matrix $X$, only CUSPARSE_OPERATION_NON_TRANSPOSE and 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 $\text{csrsm2\_bufferSize()}$. The address of $\text{pBuffer}$ must be multiple of 128 bytes. If it is not, CUSPARSE\_STATUS\_INVALID\_VALUE is returned.

Although $\text{csrsm2\_solve()}$ can be done without level information, the user still needs to be aware of consistency. If $\text{csrsm2\_analysis()}$ is called with policy CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL, $\text{csrsm2\_solve()}$ can be run with or without levels. On the contrary, if $\text{csrsm2\_analysis()}$ is called with CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL, $\text{csrsm2\_solve()}$ 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 $\text{csrsm2\_analysis()}$ with CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL once. Then do $\text{csrsm2\_solve()}$ 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 $\text{csrsm2\_solve()}$ reports the first numerical zero, including a structural zero. If $\text{status}$ 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 $\text{cusparseXcsrsm2\_zeroPivot()}$ to know where the numerical zero is.

$\text{csrsm2}$ provides two algorithms specified by the parameter $\text{algo}$. $\text{algo}=0$ is non-block version and $\text{algo}=1$ is block version. Non-block version is memory-bound, limited by bandwidth. Block version partitions the matrix into small tiles and applies dense 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 $\text{csrsm2}$.

The function supports the following properties if $\text{pBuffer} \neq \text{NULL}$

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

<table>
<thead>
<tr>
<th>Field</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 is non-block version; algo = 1 is block version.</td>
</tr>
<tr>
<td>transA</td>
<td>the operation ( \text{op}(A) ).</td>
</tr>
<tr>
<td>transB</td>
<td>the operation ( \text{op}(B) ).</td>
</tr>
<tr>
<td>m</td>
<td>number of rows and columns of matrix ( A ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>number of columns of matrix ( \text{op}(B) ).</td>
</tr>
<tr>
<td>nnz</td>
<td>number of nonzeros 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>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>B</td>
<td>(&lt;\text{type}&gt;) right-hand-side matrix. ( \text{op}(B) ) is of size ( m )-by-nrhs.</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of ( B ) and ( X ).</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>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is returned by ( \text{csrsm2_bufferSize} ).</td>
</tr>
</tbody>
</table>

## Output

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>(&lt;\text{type}&gt;) solution matrix, ( \text{op}(X) ) is of size ( m )-by-nrhs.</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus_t} for the description of the return status.
9.9. cusparseXcsrsm2_zeroPivot() [DEPRECATED]

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

```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**

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

**Output**

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

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* position)
```

See cusparseStatus_t for the description of the return status
int n,
int k,
int nnz,
const float* alpha,
const float* A,
int lda,
const float* cscValB,
const int* cscColPtrB,
const int* cscRowIndB,
const float* beta,
float* C,
int ldc)

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

cusparseStatus_t
cusparseCgemmi(cusparseHandle_t handle,
int m,
int n,
int k,
int nnz,
const cuComplex* alpha,
const cuComplex* A,
int lda,
const cuComplex* cscValB,
const int* cscColPtrB,
const int* cscRowIndB,
const cuComplex* beta,
cuComplex* C,
int ldc)

cusparseStatus_t
cusparseZgemmi(cusparseHandle_t handle,
int m,
int n,
int k,
int nnz,
const cuDoubleComplex* alpha,
const cuDoubleComplex* A,
int lda,
const cuDoubleComplex* cscValB,
const int* cscColPtrB,
const int* cscRowIndB,
const cuDoubleComplex* beta,
cuDoubleComplex* C,
int ldc)
This function performs the following matrix-matrix operations:

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

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 \( \text{cscValB} \), \( \text{cscColPtrB} \), and \( \text{cscRowIndB} \). \( \alpha \) 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;\text{type}&gt;) scalar used for multiplication.</td>
</tr>
<tr>
<td>A</td>
<td>array of dimensions ((\text{lda}, k)).</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of A. It must be at least ( m ).</td>
</tr>
<tr>
<td>cscValB</td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnz} (= \text{cscColPtrB}(k) - \text{cscColPtrB}(0)) ) nonzero elements of matrix B.</td>
</tr>
<tr>
<td>cscColPtrB</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>cscRowIndB</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>beta</td>
<td>(&lt;\text{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 ((\text{ldc}, n)).</td>
</tr>
<tr>
<td>ldc</td>
<td>leading dimension of C. It must be at least ( m ).</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>(&lt;\text{type}&gt;) updated array of dimensions ((\text{ldc}, n)).</td>
</tr>
</tbody>
</table>

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>csrgeam2()

```c
 cusparseStatus_t cusparseScsrgeam2_bufferSizeExt(cusparseHandle_t handle, int m, int n, const float* alpha, const cusparseMatDescr_t descrA, int nnzA, const float* csrSortedValA, const int* csrSortedRowPtrA, const int* csrSortedColIndA, const float* beta, const cusparseMatDescr_t descrB, int nnzB, const float* csrSortedValB, const int* csrSortedRowPtrB, const int* csrSortedColIndB, const cusparseMatDescr_t descrC, const float* csrSortedValC, const int* csrSortedRowPtrC, const int* csrSortedColIndC, size_t* pBufferSizeInBytes)
```

```c
 cusparseStatus_t cusparseDcsrgeam2_bufferSizeExt(cusparseHandle_t handle, int m, int n, const double* alpha, const cusparseMatDescr_t descrA, int nnzA, const double* csrSortedValA, const int* csrSortedRowPtrA, const int* csrSortedColIndA, const double* beta, const cusparseMatDescr_t descrB, int nnzB, const double* csrSortedValB, const int* csrSortedRowPtrB, const int* csrSortedColIndB, const cusparseMatDescr_t descrC, const double* csrSortedValC, const int* csrSortedRowPtrC, const int* csrSortedColIndC, size_t* pBufferSizeInBytes)
```
cusparseStatus_t
cusparseCcsrgeam2_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const cuComplex* alpha,
const cusparseMatDescr_t descrA,
int nnzA,
const cuComplex* csrSortedValA,
const int* csrSortedRowPtrA,
const int* csrSortedColIndA,
const cuComplex* beta,
const cusparseMatDescr_t descrB,
int nnzB,
const cuComplex* csrSortedValB,
const int* csrSortedRowPtrB,
const int* csrSortedColIndB,
const cuComplex* csrSortedValC,
const int* csrSortedRowPtrC,
const int* csrSortedColIndC,
size_t* pBufferSizeInBytes)

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

cusparseStatus_t
cusparseXcsrgeam2Nnz(cusparseHandle_t handle,
int m,
int n,
const cusparseMatDescr_t descrA,
int nnzA,
csrSortedRowPtrA,
csrSortedColIndA,
const cusparseMatDescr_t descrB,
int nnzB,
cusparseStatus_t

cusparseScsrgeam2(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)

cusparseStatus_t

cusparseDcsrgeam2(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)

cusparseStatus_t

cusparseCcsrgeam2(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

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

where \( A, B, \) and \( C \) are \( m \times n \) sparse matrices (defined in CSR storage format by the three arrays \( \text{csrValA, csrValB, csrValC, csrRowPtrA, csrRowPtrB, csrRowPtrC, and csrColIndA, csrColIndB, csrColIndC} \) 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 \( \text{csrValC, csrColIndC} \) 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,
```
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 $A$ 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

| total number of nonzero elements in device or host memory. It is equal to (csrRowPtrC(m) - 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,
k,
const float*             alpha,
const cusparseMatDescr_t descrA,
int                      nnzA,
csrRowPtrA,          csrColIndA,
const int*               csrRowPtrA,
const int*               csrColIndA,
const int*               csrRowPtrB,
const int*               csrColIndB,
const float*             beta,
csrRowPtrB,          csrColIndB,
const int*               csrRowPtrB,
const int*               csrColIndB,
csrgemm2Info_t           info,
size_t*                  pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrgemm2_bufferSizeExt(cusparseHandle_t         handle,
int                      m,
int                      n,
k,
const double*            alpha,
const cusparseMatDescr_t descrA,
int                      nnzA,
csrRowPtrA,          csrColIndA,
const int*               csrRowPtrA,
const int*               csrColIndA,
const int*               csrRowPtrB,
const int*               csrColIndB,
const double*            beta,
csrRowPtrB,          csrColIndB,
const int*               csrRowPtrB,
const int*               csrColIndB,
csrgemm2Info_t           info,
size_t*                  pBufferSizeInBytes)
```
```c
 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)
```

```c
 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)
```

```c
 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,
     csrgemm2Info_t info,
     size_t* pBufferSizeInBytes)
```
```c
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 descrC,
int nnzC,
const 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 descrC,
int nnzC,
const 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 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                      nnzC,
const cuDoubleComplex*   csrValC,
const int*               csrRowPtrC,
const int*               csrColIndC,
const csrgemm2Info_t     info,
void*                    pBuffer)
This function performs following matrix-matrix operation:

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

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 \( \text{csrValA}, \text{csrValB}, \text{csrValD}, \text{csrValC}, \text{csrRowPtrA}, \text{csrRowPtrB}, \text{csrRowPtrD}, \text{csrRowPtrC}, \text{csrColIndA}, \text{csrColIndB}, \text{csrColIndD}, \text{csrColIndC} \)) 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 A ), D is not used</td>
</tr>
<tr>
<td>!NULL</td>
<td>!NULL</td>
<td>( C = \alpha A + \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 \( A \) 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 ) is ( 0 ) or ( n ) is ( 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; ( 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 = \alpha A ) if ( \alpha ) is zero.</td>
</tr>
<tr>
<td></td>
<td>( C = \alpha A + \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 \( \text{cuSPARSE} \) 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 \( \text{nnzC} = \text{nnzTotalDevHostPtr} \) or \( \text{nnzC} = \text{csrRowPtrC(m)} - \text{csrRowPtrC(0)} \) 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 \( C = -A \times B + D \) is as follows:

```c
// assume matrices A, B and D are ready.
```
```c
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
cusparseCreateCsrmm2Info(&info);

// step 2: allocate buffer for csrmm2Nnz and csrmm2
cusparseDcsrmm2_bufferSizeExt(handle, m, n, k, &alpha,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    descrC, nnzC, csrRowPtrC, csrColIndC,
    info,
    &bufferSize);
cudamalloc(&buffer, bufferSize);

// step 3: compute csrRowPtrC
 cusparseXcsrmm2Nnz(handle, m, n, k,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    descrC, nnzC, csrRowPtrC, csrColIndC,
    descrD, nnzD, csrRowPtrD, csrColIndD,
    descrE, nnzE, csrRowPtrE, csrColIndE,
    descrF, nnzF, csrRowPtrF, csrColIndF,
    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
cusparseDcsrmm2(handle, m, n, k, &alpha,
    descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrValB, csrRowPtrB, csrColIndB,
    &beta,
    descrC, nnzC, csrValC, csrRowPtrC, csrColIndC,
    descrD, nnzD, csrValD, csrRowPtrD, csrColIndD,
    descrE, nnzE, csrValE, csrRowPtrE, csrColIndE,
    descrF, nnzF, csrValF, csrRowPtrF, csrColIndF,
    info, buffer);

// step 5: destroy the opaque structure
cusparseDestroyCsrmm2Info(info);

Several comments on csrm2() :

- 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()` supports 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>
</tbody>
</table>
### descrC
The descriptor of matrix C. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL` only.

### info
Structure with information used in `csrgemm2Nnz` and `csrgemm2`.

### pBuffer
Buffer allocated by the user; the size is returned by `csrgemm2_bufferSizeExt`.

### Output

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrValC</td>
<td><code>&lt;type&gt;</code> array of <code>nnzC</code> nonzero elements of matrix C.</td>
</tr>
<tr>
<td>csrRowPtrC</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>csrColIndC</td>
<td>Integer array of <code>nnzC</code> column indices of the nonzero elements of matrix C.</td>
</tr>
<tr>
<td>pBufferSizeInBytes</td>
<td>Number of bytes of the buffer used in <code>csrgemm2Nnz</code> and <code>csrgemm2</code>.</td>
</tr>
<tr>
<td>nnzTotalDevHostPtr</td>
<td>Total number of nonzero elements in device or host memory. It is equal to <code>(csrRowPtrC(m) - csrRowPtrC(0))</code>.</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)

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

cusparseStatus_t cusparseCcsric02_bufferSize(cusparseHandle_t handle,
                                           int m,
                                           int nnz,
                                           const cusparseMatDescr_t descrA,
                                           cuComplex* csrValA,
                                           const int* csrRowPtrA,
                                           const int* csrColIndA,
                                           csric02Info_t info,
                                           int* pBufferSizeInBytes)
```
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>\text{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>
</tbody>
</table>

**Output**
<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 csric02_analysis() and csric02().</td>
</tr>
</tbody>
</table>

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

### 11.1.2. cusparse<t>csric02_analysis()

```c

cusparseStatus_t
cusparseScsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const float* csrValA,
                          const int* csrRowPtrA,
                          const int* csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void* pBuffer)

cusparseStatus_t
cusparseDcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const double* csrValA,
                          const int* csrRowPtrA,
                          const int* csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void* pBuffer)

cusparseStatus_t
cusparseCcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const cuComplex* csrValA,
                          const int* csrRowPtrA,
                          const int* csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void* pBuffer)

cusparseStatus_t
cusparseZcsric02_analysis(cusparseHandle_t handle,
                          int m,
                          int nnz,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex* csrValA,
                          const int* csrRowPtrA,
                          const int* csrColIndA,
                          csric02Info_t info,
                          cusparseSolvePolicy_t policy,
                          void* pBuffer)
```
This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

\[ A = LL^T \]

\( A \) 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 a buffer size returned by \( \text{csric02_bufferSize()} \). The address of \( \text{pBuffer} \) must be multiple of 128 bytes. If not, \( \text{CUSPARSE_STATUS_INVALID_VALUE} \) is returned.

Function \( \text{csric02_analysis()} \) reports a structural zero and computes level information stored in the opaque structure \( \text{info} \). The level information can extract more parallelism during incomplete Cholesky factorization. However \( \text{csric02()} \) can be done without level information. To disable level information, the user must specify the policy of \( \text{csric02_analysis()} \) and \( \text{csric02()} \) as \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \). Function \( \text{csric02_analysis()} \) always reports the first structural zero, even if the policy is \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \). The user needs to call \( \text{cusparseXcsric02_zeroPivot()} \) to know where the structural zero is.

It is the user’s choice whether to call \( \text{csric02()} \) if \( \text{csric02_analysis()} \) reports a structural zero. In this case, the user can still call \( \text{csric02()} \), 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

**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; \text{ array of } \text{nzn}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0))\text{ nonzero elements of matrix } A).</td>
</tr>
<tr>
<td>csrRowPtrA</td>
<td>( \text{integer array of } m + 1 \text{ elements that contains the start of every row and the end of the last row plus one.} )</td>
</tr>
<tr>
<td>csrColIndA</td>
<td>( \text{integer array of } \text{nzn}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) \text{ column indices of the nonzero elements of matrix } A. )</td>
</tr>
<tr>
<td>info</td>
<td>structure initialized using ( \text{cusparseCreateCsric02Info()} ).</td>
</tr>
</tbody>
</table>
The supported policies are
CUSPARSE_SOLVE_POLICY_NO_LEVEL and
CUSPARSE_SOLVE_POLICY_USE_LEVEL.

| pBuffer     | buffer allocated by the user; the size is returned by csric02_bufferSize(). |

**Output**

| info        | number of bytes of the buffer used in csric02_analysis() and csric02(). |

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)
```

```c
cusparseStatus_t
cusparseDcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 double* csrValA_valM,
                 const int* csrRowPtrA,
                 const int* csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void* pBuffer)
```

```c
cusparseStatus_t
cusparseCcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 cuComplex* csrValA_valM,
                 const int* csrRowPtrA,
                 const int* csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void* pBuffer)
```

```c
cusparseStatus_t
cusparseZcsric02(cusparseHandle_t handle,
                 int m,
                 int nnz,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex* csrValA_valM,
                 const int* csrRowPtrA,
                 const int* csrColIndA,
                 csric02Info_t info,
                 cusparseSolvePolicy_t policy,
                 void* pBuffer)
```
This function performs the solve phase of the computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

\[ A \approx LL^H \]

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

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

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

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

**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 \( M \cdot y = x \) 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 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;
csrco2Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
```
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;

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

// step 2: create a empty info structure
// we need one info for csric02 and two info's for csrsv2
cusparseCreateCsric02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_Lt);

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

cusparseDcsric02_bufferSize(handle, m, nnz, descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &bufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz, descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_Lt, m, nnz, descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_Lt, pBufferSize_Lt);
pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

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

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// we can do analysis of csric02 and csrsv2 simultaneously.
cusparseCsric02_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\n", structural_zero, structural_zero);
}
cusparseCsrsv2_analysis(handle, trans_L, m, nnz, descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, policy_L, pBuffer);

cusparseCsrsv2_analysis(handle, trans_Lt, m, nnz, descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_Lt, policy_Lt, pBuffer);

// step 5: M = L * L'
cusparseCsric02(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>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(ceil 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(ceil 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>
cuSPARSE Preconditioners Reference

| pBuffer | buffer allocated by the user; the size is returned by csric02_bufferSize(). |

Output

| csrValA_valM | <type> matrix containing the incomplete-Cholesky lower triangular factor. |

See cusparseStatus_t for the description of the return status

### 11.1.4. cusparseXcsric02_zeroPivot()

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

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

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

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

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

- 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 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()

cusparseStatus_t cusparseSbsric02_bufferSize(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, cusparseDirectIon_t int)
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 \( bsrValA \), \( bsrRowPtrA \), and \( bsrColIndA \).

The buffer size depends on the dimensions of \( mb \), \( blockDim \), and the number of nonzero blocks of the matrix \( nnzb \). If the user changes the matrix, it is necessary to call \( bsric02_bufferSize() \) again to have the correct buffer size; otherwise, a segmentation fault may occur.
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>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 (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>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Parameter</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 bsric02_analysis() and bsric02().</td>
</tr>
</tbody>
</table>

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, const cusparseMatDescr_t descrA, const float* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, int info, int pBufferSizeInBytes)
```
This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

\[ A = LL^H \]

\( A \) is an \((mb*blockDim)x(mb*blockDim)\) sparse matrix that is defined in BSR storage format by the three arrays \( bsrValA, bsrRowPtrA, \) and \( bsrColIndA \). The block in BSR format is of size \( blockDim*blockDim \), stored as column-major or row-major as determined by parameter \( dirA \), which is either \texttt{CUSPARSE_DIRECTION_COLUMN} or \texttt{CUSPARSE_DIRECTION_ROW}. The matrix type must be \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}, and the fill mode and diagonal type are ignored.
This function requires a buffer size returned by \texttt{bsric02\_bufferSize90}. The address of \texttt{pBuffer} must be a multiple of 128 bytes. If it is not, \texttt{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

Function \texttt{bsric02\_analysis()} reports structural zero and computes level information stored in the opaque structure \texttt{info}. The level information can extract more parallelism during incomplete Cholesky factorization. However \texttt{bsric02()} can be done without level information. To disable level information, the user needs to specify the parameter \texttt{policy} of \texttt{bsric02[\_analysis|]} as \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}.

Function \texttt{bsric02\_analysis} always reports the first structural zero, even when parameter \texttt{policy} is \texttt{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}. The user must call \texttt{cusparseXbsric02\_zeroPivot()} to know where the structural zero is.

It is the user’s choice whether to call \texttt{bsric02()} if \texttt{bsric02\_analysis()} reports a structural zero. In this case, the user can still call \texttt{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 \textbf{not} support asynchronous execution
- The routine does \textbf{not} support 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{dirA}</td>
<td>storage format of blocks, either \texttt{CUSPARSE_DIRECTION_ROW} or \texttt{CUSPARSE_DIRECTION_COLUMN}.</td>
</tr>
<tr>
<td>\texttt{mb}</td>
<td>number of block rows and block columns of matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{nnzb}</td>
<td>number of nonzero blocks of matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{descrA}</td>
<td>the descriptor of matrix \texttt{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{bsrValA}</td>
<td>\texttt{&lt;type&gt;} array of \texttt{nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0))} nonzero blocks of matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{bsrRowPtrA}</td>
<td>integer array of \texttt{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>\texttt{bsrColIndA}</td>
<td>integer array of \texttt{nnzb (= bsrRowPtrA(mb) - bsrRowPtrA(0))} column indices of the nonzero blocks of matrix \texttt{A}.</td>
</tr>
<tr>
<td>\texttt{blockDim}</td>
<td>block dimension of sparse matrix \texttt{A}; must be larger than zero.</td>
</tr>
<tr>
<td>\texttt{info}</td>
<td>structure initialized using \texttt{cusparseCreateBsric02Info()}.</td>
</tr>
</tbody>
</table>
### 11.1.7. `cusparse<t>bsric02()`

**Input**

<table>
<thead>
<tr>
<th><strong>policy</strong></th>
<th>the supported policies are <code>CUSPARSE_SOLVE_POLICY_NO_LEVEL</code> and <code>CUSPARSE_SOLVE_POLICY_USE_LEVEL</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pBuffer</strong></td>
<td>buffer allocated by the user; the size is returned by <code>bsric02_bufferSize()</code>.</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.
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 \((mb*\text{blockDim}) \times (mb*\text{blockDim})\) 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 \( \text{blockDim} \times \text{blockDim} \), 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.

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

Although \( \text{bsric02()} \) can be done without level information, the user must be aware of consistency. If \( \text{bsric02\_analysis()} \) is called with policy \text{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL}, \( \text{bsric02()} \) can be run with or without levels. On the other hand, if \( \text{bsric02\_analysis()} \) is called with \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}, \( \text{bsric02()} \) can only accept \text{CUSPARSE\_SOLVE\_POLICY\_NO\_LEVEL}; otherwise, \text{CUSPARSE\_STATUS\_INVALID\_VALUE} is returned.

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

Function \( \text{bsric02()} \) reports the first numerical zero, including a structural zero. The user must call \( \text{cusparseXbsric02\_zeroPivot()} \) to know where the numerical zero is.

The \( \text{bsric02()} \) 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 \( \text{bsric02()} \), \( A \) 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 \( m=mb*\text{blockDim} \). The following code solves precondition system \( M^\ast y = x \), where \( M \) is the product of Cholesky factorization \( L \) and its transpose.


\[ M = LL^H \]

// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
bsric02Info_t info_M = 0;
bsrsv2Info_t info_L = 0;
bsrsv2Info_t info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.0f;
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);

// step 2: create a empty info structure
// we need one info for bsric02 and two info's for bsrsv2

cusparseCreateBsric02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_Lt);

// step 3: query how much memory used in bsric02 and bsrsv2, and allocate the buffer

cusparseDbsric02_bufferSize(handle, dir, mb, nnzb,
  descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M, &bufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
  descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L, &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_Lt, mb, nnzb,
  descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_Lt, &pBufferSize_Lt);

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

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

// step 4: perform analysis of incomplete Cholesky on M
// perform analysis of triangular solve on L
// perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// 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, infoLt,
    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(infoLt);
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</td>
</tr>
<tr>
<td></td>
<td>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>----------</td>
<td>------------------------------------------------------------</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 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 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>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is returned by bsric02_bufferSize().</td>
</tr>
</tbody>
</table>

**Output**

| bsrValA | <type> matrix containing the incomplete-Cholesky lower triangular factor. |

See [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 CUSPARSE_STATUS_ZERO_PIVOT, position=j means A(j,j) has either a structural zero or a numerical zero (the block is not positive definite). Otherwise position=-1.

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

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

- 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>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>info contains a structural zero or a numerical zero if the user already called <code>bsric02_analysis()</code> or <code>bsric02()</code>.</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>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.</td>
</tr>
</tbody>
</table>

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()`

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

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

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

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

```c
if (tol >= fabs(A(j,j))) then A(j,j) = boost_val.
```

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

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

Both `tol` and `boost_val` can be in the host memory or device memory. The user can set the proper mode with `cusparseSetPointerMode()`.

- 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>info</code></td>
<td>structure initialized using <code>cusparseCreateCsrilu02Info()</code>.</td>
</tr>
<tr>
<td><code>enable_boost</code></td>
<td>disable boost by <code>enable_boost=0</code>; otherwise, boost is enabled.</td>
</tr>
<tr>
<td><code>tol</code></td>
<td>tolerance to determine a numerical zero.</td>
</tr>
<tr>
<td><code>boost_val</code></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()`

```c
cusparseStatus_t cusparseScsrilu02_bufferSize(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, float* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, int* pBufferSizeInBytes);
```

```c
cusparseStatus_t cusparseDcsrilu02_bufferSize(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, double* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, int* pBufferSizeInBytes);
```
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 \( nnz \), 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>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</td>
<td>(&lt;\text{type}&gt;) array of ( 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 \(nnz(=csrRowPtrA(m) - csrRowPtrA(0))\) column indices of the nonzero elements of matrix \(A\).

##Output

###info
record internal states based on different algorithms.

###pBufferSizeInBytes
number of bytes of the buffer used in `csrilu02_analysis()` and `csrilu02()`.

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

###11.2.3. `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)

cusparseStatus_t cusparseDcsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const double* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)

cusparseStatus_t cusparseCcsrilu02_analysis(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, const cuComplex* csrValA, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)

cusparseStatus_t cusparseZcsrilu02_analysis(cusparseHandle_t handle, 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}(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>
### 11.2.4. `cusparse<t>csrilu02()`

The `cusparse<t>csrilu02()` function is used to compute the preconditioner for the iterative solver.

**Input**

- **info**: structure initialized using `cusparseCreateCsrilu02Info()`.
- **policy**: the supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.
- **pBuffer**: buffer allocated by the user, the size is returned by `csrilu02_bufferSize()`.

**Output**

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

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

The function signatures are as follows:

- **`cusparseScsrilu02()`**

  ```c
  cusparseStatus_t cusparseScsrilu02(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, float* csrValA_valM, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)
  ```

- **`cusparseDcsrilu02()`**

  ```c
  cusparseStatus_t cusparseDcsrilu02(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, double* csrValA_valM, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)
  ```

- **`cusparseCcsrilu02()`**

  ```c
  cusparseStatus_t cusparseCcsrilu02(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, cuComplex* csrValA_valM, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)
  ```

- **`cusparseZcsrilu02()`**

  ```c
  cusparseStatus_t cusparseZcsrilu02(cusparseHandle_t handle, int m, int nnz, const cusparseMatDescr_t descrA, cuDoubleComplex* csrValA_valM, const int* csrRowPtrA, const int* csrColIndA, csrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)
  ```
This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

\[ A ≈ LU \]

\( A \) 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 \( \text{csrilu02} \_\text{bufferSize}() \). 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 \( \text{csrilu02} \_\text{analysis}() \) is called with policy \( \text{CUSPARSE\_SOLVE\_POLICY\_USE\_LEVEL} \), \( \text{csrilu02}() \) can be run with or without levels. On the other hand, if \( \text{csrilu02} \_\text{analysis}() \) 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 \( \text{cusparseXcsrilu02} \_\text{zeroPivot}() \) 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;
in 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 an 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,
// step 5: M = L * U
cusparseDcsrilu02(handle, m, nnz, descr_M,
    d_csrVal, d_csrRowPtr, d_csrColInd, info_M, policy_M, pBuffer);
status = cusparseXcsrilu02_zeroPivot(handle, info_M, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
    printf("U(%d,%d) is zero\n", numerical_zero, numerical_zero);
}

// step 6: solve L*z = x
cusparseDcsrsv2_solve(handle, trans_L, m, nnz, &alpha, descr_L,
    d_csrVal, d_csrRowPtr, d_csrColInd, info_L,
    d_x, d_z, policy_L, pBuffer);

// step 7: solve U*y = z
cusparseDcsrsv2_solve(handle, trans_U, m, nnz, &alpha, descr_U,
    d_csrVal, d_csrRowPtr, d_csrColInd, info_U,
    d_z, d_y, policy_U, pBuffer);

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyMatDescr(descr_M);
cusparseDestroyMatDescr(descr_L);
cusparseDestroyMatDescr(descr_U);
cusparseDestroyCsrilu02Info(info_M);
cusparseDestroyCsrsv2Info(info_L);
cusparseDestroyCsrsv2Info(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>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>
policy | the supported policies are CUSPARSE_SOLVE_POLICY_NO_LEVEL and CUSPARSE_SOLVE_POLICY_USE_LEVEL.
--- | ---
pBuffer | buffer allocated by the user; the size is returned by csrilu02_bufferSize().

Output

| csrValA_valM | <type> matrix containing the incomplete-LU lower and upper triangular factors. |

See `cusparseStatus_t` for the description of the return status

### 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()`
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><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>info</code></td>
<td>structure initialized using <code>cusparseCreateBsrilu02Info()</code>.</td>
</tr>
<tr>
<td><code>enable_boost</code></td>
<td>disable boost by setting <code>enable_boost=0</code>. Otherwise, boost is enabled.</td>
</tr>
<tr>
<td><code>tol</code></td>
<td>tolerance to determine a numerical zero.</td>
</tr>
<tr>
<td><code>boost_val</code></td>
<td>boost value to replace a numerical zero.</td>
</tr>
</tbody>
</table>
11.2.7. `cusparse<t>bsrilu02_bufferSize()`

This function returns the size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting.

See `cusparseStatus_t` for the description of the return status.
$A = LU$

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

The buffer size depends on the dimensions of $mb$, $blockDim$, and the number of nonzero blocks of the matrix $nnzb$. If the user changes the matrix, it is necessary to call $bsrilu02\_bufferSize()$ again to have the correct buffer size; otherwise, a segmentation fault may occur.

**Input**

| handle | handle to the cuSPARSE library context. |
| dirA | storage format of blocks, either CUSPARSE\_DIRECTION\_ROW or CUSPARSE\_DIRECTION\_COLUMN. |
| mb | number of block rows and columns of matrix $A$. |
| nnzb | number of nonzero blocks of matrix $A$. |
| descrA | the descriptor of matrix $A$. The supported matrix type is CUSPARSE\_MATRIX\_TYPE\_GENERAL. Also, the supported index bases are CUSPARSE\_INDEX\_BASE\_ZERO and CUSPARSE\_INDEX\_BASE\_ONE. |
| bsrValA | $<type>$ array of $nnzb = (bsrRowPtrA(mb) - bsrRowPtrA(0))$ nonzero blocks of matrix $A$. |
| bsrRowPtrA | integer array of $mb + 1$ elements that contains the start of every block row and the end of the last block row plus one. |
| bsrColIndA | integer array of $nnzb = (bsrRowPtrA(mb) - bsrRowPtrA(0))$ column indices of the nonzero blocks of matrix $A$. |
| blockDim | block dimension of sparse matrix $A$, larger than zero. |

**Output**

| info | record internal states based on different algorithms. |
| pBufferSizeInBytes | number of bytes of the buffer used in $bsrilu02\_analysis()$ and $bsrilu02()$. |

**Status Returned**

- CUSPARSE\_STATUS\_SUCCESS: the operation completed successfully.
- CUSPARSE\_STATUS\_NOT\_INITIALIZED: the library was not initialized.
- CUSPARSE\_STATUS\_ALLOC\_FAILED: the resources could not be allocated.
- CUSPARSE\_STATUS\_INVALID\_VALUE: invalid parameters were passed ($mb$, $nnzb$)$\leq 0$, base index is not 0 or 1.
- CUSPARSE\_STATUS\_ARCH\_MISMATCH: the device only supports compute capability 2.0 and above.
### 11.2.8. `cusparse<t>bsrilu02_analysis()`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cusparseStatus_t cusparseSbsrilu02_analysis(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, int nnzb, const cusparseMatDescr_t descrA, float* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, bsrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseStatus_t cusparseDbsrilu02_analysis(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, int nnzb, const cusparseMatDescr_t descrA, double* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, bsrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseStatus_t cusparseCbsrilu02_analysis(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, int nnzb, const cusparseMatDescr_t descrA, cuComplex* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, bsrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseStatus_t cusparseZbsrilu02_analysis(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, int nnzb, const cusparseMatDescr_t descrA, cuDoubleComplex* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, bsrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)</code></td>
<td></td>
</tr>
</tbody>
</table>
This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting

\[ A \approx LU \]

\( A \) is an \((mb \times blockDim) \times (mb \times blockDim)\) 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 \texttt{CUSPARSE_DIRECTION_COLUMN} or \texttt{CUSPARSE_DIRECTION_ROW}. The matrix type must be \texttt{CUSPARSE_MATRIX_TYPE_GENERAL}, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by \texttt{bsrilu02_bufferSize()} . The address of \( pBuffer \) must be multiple of 128 bytes. If it is not, \texttt{CUSPARSE_STATUS_INVALID_VALUE} is returned.

Function \texttt{bsrilu02_analysis()} reports a structural zero and computes level information stored in the opaque structure \( info \). The level information can extract more parallelism during incomplete LU factorization. However \texttt{bsrilu02()} can be done without level information. To disable level information, the user needs to specify the parameter \texttt{policy} of \texttt{bsrilu02[\_analysis\_]} as \texttt{CUSPARSE_SOLVE_POLICY_NO_LEVEL}.

Function \texttt{bsrilu02_analysis()} always reports the first structural zero, even with parameter \texttt{policy} is \texttt{CUSPARSE_SOLVE_POLICY_NO_LEVEL}. The user must call \texttt{cusparseXbsrilu02_zeroPivot()} to know where the structural zero is.

It is the user’s choice whether to call \texttt{bsrilu02()} if \texttt{bsrilu02_analysis()} reports a structural zero. In this case, the user can still call \texttt{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 \textbf{not} support asynchronous execution
- The routine does \textbf{not} support CUDA graph capture

\begin{itemize}
  \item Input
\end{itemize}

\begin{verbatim}

type                        descrA
<type> array of nnzb ( = bsrRowPtrA(mb) - bsrRowPtrA(0) ) nonzero blocks of matrix A.
\end{verbatim}
### 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 \( \text{nnzb} = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0) \) column indices of the nonzero blocks of matrix \( A \).

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

### info
structure initialized using \( \text{cusparseCreateBsrilu02Info()} \).

### policy
the supported policies are \( \text{CUSPARSE_SOLVE_POLICY_NO_LEVEL} \) and \( \text{CUSPARSE_SOLVE_POLICY_USE_LEVEL} \).

### pBuffer
buffer allocated by the user, the size is returned by \( \text{bsrilu02_bufferSize()} \).

### 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 cusparseDbsrilu02(cusparseHandle_t handle, cusparseDirection_t dirA, int mb, int nnzb, const cusparseMatDescr_t descry, double* bsrValA, const int* bsrRowPtrA, const int* bsrColIndA, int blockDim, bsrilu02Info_t info, cusparseSolvePolicy_t policy, void* pBuffer)
```
This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting

\[ A \approx LU \]

\( A \) is an \((mb \times blockDim) \times (mb \times blockDim)\) 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 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, \( \text{bsr2csr(bsrilu02(A))} = \text{csrilu02(bsr2csr(A))} \). 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 $m$-by-$m$ 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 $U$.

```
// Suppose that $A$ is a $m \times m$ 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);
cusparseDbsrv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L, &pBufferSize_L);
cusparseDbsrv2_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);
}
cusparseDbsrv2_analysis(handle, dir, trans_L, mb, nnzb, descr_L,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
    policy_L, pBuffer);
cusparseDbsrv2_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
cusparseDbsrv2_solve(handle, dir, trans_L, mb, nnzb, descr_L,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
    d_x, d_z, policy_L, pBuffer);

// step 7: solve U*y = z
cusparseDbsrv2_solve(handle, dir, trans_U, mb, nnzb, descr_U,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_U,
    d_y, d_z, policy_U, pBuffer);

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyMatDescr(descr_M);
cusparseDestroyMatDescr(descr_L);
cusparseDestroyMatDescr(descr_U);
cusparseDestroySrilu02Info(info_M);
cusparseDestroySrv2Info(info_L);
cusparseDestroySrv2Info(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 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 $nnzb = \text{bsrRowPtrA}(mb) - \text{bsrRowPtrA}(0)$ 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 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>
<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()

```c

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>
</table>
| info                  | info contains structural zero or numerical zero if the user already called `bsrilu02_analysis()` or `bsrilu02()`.

**Output**

| position              | if no structural or numerical zero, `position` is -1; otherwise if $A(j,j)$ is missing or $U(j,j)$ is not invertible, `position=j`. |

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 cusparseGtadv2_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)
```
const double* d,
const double* du,
const double* B,
int ldb,
size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const cuComplex* dl,
const cuComplex* d,
const cuComplex* du,
const cuComplex* B,
int ldb,
size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseZgtsv2_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const cuDoubleComplex* dl,
const cuDoubleComplex* d,
const cuDoubleComplex* du,
const cuDoubleComplex* B,
int ldb,
size_t* bufferSizeInBytes)

This function returns the size of the buffer used in gtsv2 which computes the solution of a tridiagonal linear system with multiple right-hand sides.

\[ A \times X = B \]

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>
</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()

cusparseStatus_t
cusparseSgtsv2(cusparseHandle_t handle,
    int       m,
    int       n,
    const float*    dl,
    const float*    d,
    const float*    du,
    float*              B,
    int            ldb,
    void             pBuffer)

cusparseStatus_t
cusparseDgtsv2(cusparseHandle_t handle,
    int       m,
    int       n,
    const double*    dl,
    const double*    d,
    const double*    du,
    double*              B,
    int            ldb,
    void             pBuffer)

cusparseStatus_t
cusparseCgtsv2(cusparseHandle_t handle,
    int       m,
    int       n,
    const cuComplex* dl,
    const cuComplex* d,
    const cuComplex* du,
    cuComplex*       B,
    int            ldb,
    void             pBuffer)

cusparseStatus_t
cusparseZgtsv2(cusparseHandle_t handle,
    int       m,
    int       n,
    const cuDoubleComplex* dl,
    const cuDoubleComplex* d,
    const cuDoubleComplex* du,
    cuDoubleComplex*       B,
    int            ldb,
    void             pBuffer)

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:
\[
A \times X = B
\]

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.

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 routine does perform pivoting, which usually results in more accurate and more stable results than \texttt{cusparse\textless t\textgreater gtsv\_nopivot()} or \texttt{cusparse\textless t\textgreater gtsv2\_nopivot()} at the expense of some execution time.

This function requires a buffer size returned by \texttt{gtsv2\_bufferSizeExt()} . The address of \(pBuffer\) 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>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>the size of the linear system ([\text{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>
<tr>
<td>B</td>
<td>(&lt;\text{type}&gt;) dense right-hand-side array of dimensions ((\text{ldb}, n)).</td>
</tr>
<tr>
<td>ldb</td>
<td>leading dimension of (B) ([\text{that is } \geq \text{max}(1, m)]).</td>
</tr>
<tr>
<td>pBuffer</td>
<td>buffer allocated by the user, the size is return by \texttt{gtsv2_bufferSizeExt}.</td>
</tr>
</tbody>
</table>

**Output**
B
<type> dense solution array of dimensions (ldb, n).

See <code>cusparseStatus_t</code> for the description of the return status

### 11.3.3. <code>cusparse&lt;t&gt;gtsv2_nopivot_bufferSizeExt()</code>

```c
 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)
```

```c
 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)
```

```c
 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)
```

```c
 cusparseStatus_t
cusparseZgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
    int m,
    int n,
    const cuDoubleComplex* dl,
    const cuDoubleComplex* d,
    const cuDoubleComplex* du,
    const cuDoubleComplex* B,
    int ldb,
    size_t* bufferSizeInBytes)
```

This function returns the size of the buffer used in <code>gtsv2_nopivot</code> 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>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

**Output**

| \( \text{pBufferSizeInBytes} \) | number of bytes of the buffer used in the \( \text{gtsv2\_nopivot} \). |

See **cusparseStatus\_t** for the description of the return status.

### 11.3.4. cusparse\( <\text{t}>\)gtsv2\_nopivot()

```c

/* Usage Example */
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)

/* Usage Example */
cusparseStatus_t
 cusparseDgtsv2\_nopivot (cusparseHandle_t handle,
                 int m,
                 int n,
                 const double* dl,
```
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 \( \text{pBuffer} \) 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>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>
### 11.4. Batched Tridiagonal Solve

Different algorithms for batched tridiagonal solve are discussed in this section.

#### 11.4.1. `cusparse<t>gtsv2StridedBatch_bufferSizeExt()`

```c
cusparseStatus_t
cusparseSgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
int m,
const float* dl,
const float* d,
const float* du,
const float* x,
int batchCount,
int batchStride,
size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseDgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
int m,
const double* dl,
const double* d,
const double* du,
const double* x,
int batchCount,
int batchStride,
size_t* bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
int m,
const float* dl,
const float* d,
const float* du,
const float* x,
int batchCount,
int batchStride,
size_t* bufferSizeInBytes)
```
This function returns the size of the buffer used in `gtsv2StridedBatch` which computes the solution of multiple tridiagonal linear systems for $i=0,\ldots,\text{batchCount}$:

$$A^{(i)} \cdot y^{(i)} = x^{(i)}$$

The coefficient matrix $A$ of each of these tridiagonal 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 $\text{batchStride}$ in memory.

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

**Input**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>n</td>
<td>the size of the linear system $n$ (must be $\geq 3$).</td>
</tr>
<tr>
<td>dl</td>
<td><code>&lt;type&gt;</code> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $d^{(i)}$ 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>d</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>du</td>
<td><code>&lt;type&gt;</code> 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>
</tbody>
</table>
The last element of each upper diagonal must be zero. Also, the right-hand-side of the tri-diagonal linear system starts at location \(du+batchStride\times i\) in memory. The right-hand-side \(x^0\) corresponds to the \(i\)th linear system starts at location \(x+batchStride\times i\) in memory.

**x**

A \(<\text{type}>\) dense array that contains the right-hand-side of the tri-diagonal linear system.

**batchCount**

Number of systems to solve.

**batchStride**

Stride (number of elements) that separates the vectors of every system (must be at least \(m\)).

**pBufferSizeInBytes**

Number of bytes of the buffer used in the \(\text{gtsv2StridedBatch}\).

See **\text{cusparseStatus_t}** for the description of the return status.

### 11.4.2. \text{cusparse}<\text{t}>\text{gtsv2StridedBatch()}\)

```c
 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 \( i=0, \ldots, \text{batchCount} \):

\[
A^{(i)} \cdot y^{(i)} = x^{(i)}
\]

The coefficient matrix \( A \) of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (\( dl \)), 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 \( \text{batchStride} \) 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 \( \text{gtsv2StridedBatch\_bufferSizeExt()} \). The address of \( \text{pBuffer} \) 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>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>n</td>
<td>the size of the linear system (must be ( \geq 3 )).</td>
</tr>
<tr>
<td>dl</td>
<td>&lt;type&gt; dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal ( d^{(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>d</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>du</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>
</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 \( \mathbf{x}^i \) 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}.

**Output**

\( x \)  
\(<\text{type}>\) dense array that contains the solution of the tri-diagonal linear system. The solution \( \mathbf{x}^i \) 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}<\text{t}>gtsvInterleavedBatch()}

\begin{verbatim}
cusparseStatus_t  
cusparseSgtsvInterleavedBatch_bufferSizeExt (cusparseHandle_t handle,  
int               algo,  
int               m,  
const float*     dl,  
const float*     d,  
const float*     du,  
const float*     x,  
int              size_t*  

\text{pBufferSizeInBytes})

cusparseStatus_t  
cusparseDgtsvInterleavedBatch_bufferSizeExt (cusparseHandle_t handle,  
int               algo,  
int               m,  
const double*    dl,  
const double*    d,  
const double*    du,  
const double*    x,  
int              size_t*  

\text{pBufferSizeInBytes})

cusparseStatus_t  
cusparseCgtsvInterleavedBatch_bufferSizeExt (cusparseHandle_t handle,  
int               algo,  
int               m,  
const cuComplex* dl,  
const cuComplex* d,  
const cuComplex* du,  
int              size_t*  

\text{pBufferSizeInBytes})
\end{verbatim}
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusparseStatus_t</td>
<td>const cuComplex* x,</td>
</tr>
<tr>
<td>cusparseZgtsvInterleavedBatch_bufferSizeExt</td>
<td>int batchCount,</td>
</tr>
<tr>
<td></td>
<td>size_t*</td>
</tr>
<tr>
<td>cusparseStatus_t</td>
<td>const cuDoubleComplex* dl,</td>
</tr>
<tr>
<td>cusparseSgtsvInterleavedBatch</td>
<td>const cuDoubleComplex* d,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* du,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* x,</td>
</tr>
<tr>
<td></td>
<td>size_t*</td>
</tr>
<tr>
<td>cusparseStatus_t</td>
<td>const cuDoubleComplex* dl,</td>
</tr>
<tr>
<td>cusparseDgtsvInterleavedBatch</td>
<td>const cuDoubleComplex* d,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* du,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* x,</td>
</tr>
<tr>
<td></td>
<td>batchCount,</td>
</tr>
<tr>
<td></td>
<td>void* pBuffer</td>
</tr>
<tr>
<td>cusparseStatus_t</td>
<td>const cuDoubleComplex* dl,</td>
</tr>
<tr>
<td>cusparseCgtsvInterleavedBatch</td>
<td>const cuDoubleComplex* d,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* du,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* x,</td>
</tr>
<tr>
<td></td>
<td>int batchCount,</td>
</tr>
<tr>
<td></td>
<td>void* pBuffer</td>
</tr>
<tr>
<td>cusparseStatus_t</td>
<td>const cuDoubleComplex* dl,</td>
</tr>
<tr>
<td>cusparseZgtsvInterleavedBatch</td>
<td>const cuDoubleComplex* d,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* du,</td>
</tr>
<tr>
<td></td>
<td>const cuDoubleComplex* x,</td>
</tr>
<tr>
<td></td>
<td>int batchCount,</td>
</tr>
<tr>
<td></td>
<td>void* pBuffer</td>
</tr>
</tbody>
</table>
This function computes the solution of multiple tridiagonal linear systems for $i=0, \ldots, \text{batchCount}$:

$$A^i \cdot x^i = b^i$$

The coefficient matrix $A$ of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower ($d_l$), 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:

- $d_l(i) := A(i, i-1)$ 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 $d_u(m) = 0$.

The data layout is different from $\text{gtsvStridedBatch}$ which aggregates all matrices one after another. Instead, $\text{gtsvInterleavedBatch}$ gathers different matrices of the same element in a continuous manner. If $d_l$ is regarded as a 2-D array of size $m$-by-$\text{batchCount}$, $d_l(:,j)$ to store $j$-th matrix. $\text{gtsvStridedBatch}$ uses column-major while $\text{gtsvInterleavedBatch}$ uses row-major.

The routine provides three different algorithms, selected by parameter $\text{algo}$. The first algorithm is $\text{cuThomas}$ provided by Barcelona Supercomputing Center. The second algorithm is LU with partial pivoting and last algorithm is QR. From stability perspective, $\text{cuThomas}$ 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 $\text{cuThomas}$.

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

If the user prepares aggregate format, one can use $\text{cublasXgeam}$ 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>handle</th>
<th>handle to the cuSPARSE library context.</th>
</tr>
</thead>
<tbody>
<tr>
<td>algo</td>
<td>algo = 0: $\text{cuThomas}$ (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>
</tbody>
</table>
### 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
cusparseSgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
 int algo,
 int m,
 const float* ds,
 const float* dl,
 const float* d,
 const float* du,
 const float* dw,
 const float* x,
 int batchCount,
 size_t* pBufferSizeInBytes)
```

```c
 cusparseStatus_t
cusparseDgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
 int algo,
 int m,
 const double* ds,
 const double* dl,
 const double* d,
 const double* du,
 const double* dw,
 const double* x,
 int batchCount,
 size_t* pBufferSizeInBytes)
```
cusparseStatus_t
cusparseCgpgsvInterleavedBatch_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
cusparseZgpgsvInterleavedBatch_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
cusparseSgpgsvInterleavedBatch (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
cusparseDgpgsvInterleavedBatch (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
cusparseCgpgsvInterleavedBatch (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 (\(ds, dl\)), main (\(d\)), and upper (\(du, dw\)) matrix diagonals; the right-hand sides are stored in the dense matrix \(B\). Notice that solution \(X\) overwrites right-hand-side matrix \(B\) on exit.

Assuming \(A\) is of size \(m\) and base-1, \(ds, dl, d, du\) and \(dw\) are defined by the following formula:

\[
ds(i) := A(i, i-2) \quad \text{for} \quad i=1,2,\ldots,m
\]

The first two elements of \(ds\) is out-of-bound \(\{ds(1) := A(1,-1), ds(2) := A(2,0)\}\), so \(ds(1) = 0\) and \(ds(2) = 0\).

\[
dl(i) := A(i, i-1) \quad \text{for} \quad i=1,2,\ldots,m
\]

The first element of \(dl\) is out-of-bound \(dl(1) := A(1,0)\), so \(dl(1) = 0\).

\[
d(i) = A(i,i) \quad \text{for} \quad i=1,2,\ldots,m
\]

\[
du(i) = A(i,i+1) \quad \text{for} \quad i=1,2,\ldots,m
\]

The last element of \(du\) is out-of-bound \(du(m) := A(m,m+1)\), so \(du(m) = 0\).

\[
dw(i) = A(i,i+2) \quad \text{for} \quad i=1,2,\ldots,m
\]

The last two elements of \(dw\) is out-of-bound \(\{dw(m-1) := A(m-1,m+1), dw(m) := A(m,m+2)\}\), so \(dw(m-1) = 0\) and \(dw(m) = 0\).

The data layout is the same as \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>Name</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><code>&lt;type&gt;</code> 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><code>&lt;type&gt;</code> 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><code>&lt;type&gt;</code> dense array containing the main diagonal of the penta-diagonal linear system.</td>
</tr>
<tr>
<td>du</td>
<td><code>&lt;type&gt;</code> 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><code>&lt;type&gt;</code> 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><code>&lt;type&gt;</code> 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 <code>gpsvInterleavedBatch_bufferSizeExt</code>.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td><code>&lt;type&gt;</code> dense solution array of dimensions (batchCount, n).</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
Chapter 12. cuSPARSE Reorderings
Reference

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

12.1. \texttt{cusparse<t>csrcolor()}
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></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 (= 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>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>csrColIndA</code></td>
<td>integer array of ( \text{nnz} = \text{csrRowPtr}(m) - \text{csrRowPtr}(0) ) column indices of the nonzero elements of matrix A.</td>
</tr>
<tr>
<td><code>fractionToColor</code></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><code>info</code></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><code>ncolors</code></td>
<td>The number of distinct colors used (at most the size of the matrix, but likely much smaller).</td>
</tr>
<tr>
<td><code>coloring</code></td>
<td>The resulting coloring permutation</td>
</tr>
<tr>
<td><code>reordering</code></td>
<td>The resulting reordering permutation (untouched if NULL)</td>
</tr>
</tbody>
</table>

See [cusparseStatus_t](#) for the description of the return status.
This chapter describes the conversion routines between different sparse and dense storage formats.

`coosort`, `csrsort`, `cscsort`, 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</td>
<td>&gt; 20*n bytes</td>
<td>100M</td>
</tr>
<tr>
<td>cscsort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>csru2csr</td>
<td>'d' &gt; 28*n bytes</td>
<td>71M for 'd' and 55M for 'z'</td>
</tr>
<tr>
<td></td>
<td>&gt; 36*n bytes</td>
<td></td>
</tr>
</tbody>
</table>

### 13.1. cusparse<t>brr2csr()

```c
 cusparseStatus_t
cusparseSbsr2csr(cusparseHandle_t handle,              
cusparseDirection_t dir, 
int mb,  
int nb,  
const cusparseMatDescr_t descrA,  
const float* bsrValA,  
const int* bsrRowPtrA,  
const int* bsrColIndA,  
const int blockDim,  
const cusparseMatDescr_t descrC,  
float* csrValC,  
int* csrRowPtrC,  
int* csrColIndC)  
```

```c
 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 \( m = mb \times blockDim \) be the number of rows of \( A \) and \( n = nb \times blockDim \) be number of columns of \( A \), then \( A \) and \( C \) are \( m \times n \) sparse matrices. The BSR format of \( A \) contains \( nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0] \) nonzero blocks, whereas the sparse matrix \( A \) contains \( nnz = nnzb \times blockDim \times blockDim \) 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
```
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>Variable</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>Variable</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()**

cusparseStatus_t  
cusparseSgebsr2gebsc_bufferSize(cusparseHandle_t handle,
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,
void* pBuffer)
cusparseStatus_t
cusparseDgebsr2gebsc(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const double* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
double* bscVal,
int* bscRowInd,
int* bscColPtr,
cusparseAction_t copyValues,
cusparseIndexBase_t baseIdx,
void* pBuffer)

cusparseStatus_t
cusparseCgebsr2gebsc(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const cuComplex* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
cuComplex* bscVal,
int* bscRowInd,
int* bscColPtr,
cusparseAction_t copyValues,
cusparseIndexBase_t baseIdx,
void* pBuffer)

cusparseStatus_t
cusparseZgebsr2gebsc(cusparseHandle_t handle,
int mb,
int nb,
int nnzb,
const cuDoubleComplex* bsrVal,
const int* bsrRowPtr,
const int* bsrColInd,
int rowBlockDim,
int colBlockDim,
cuDoubleComplex* bscVal,
int* bscRowInd,
int* bscColPtr,
cusparseAction_t copyValues,
cusparseIndexBase_t baseIdx,
void* pBuffer)

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 \( pBuffer \neq NULL \)

The routine does **not** support asynchronous execution

The routine does **not** support CUDA graph capture

### Input

<table>
<thead>
<tr>
<th>Handle</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>mb</code></td>
<td>number of block rows of sparse matrix ( A ).</td>
</tr>
<tr>
<td><code>nb</code></td>
<td>number of block columns of sparse matrix ( A ).</td>
</tr>
<tr>
<td><code>nnzb</code></td>
<td>number of nonzero blocks of matrix ( A ).</td>
</tr>
<tr>
<td><code>bsrVal</code></td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnzb} \times \text{rowBlockDim} \times \text{colBlockDim} ) nonzero elements of matrix ( A ).</td>
</tr>
<tr>
<td><code>bsrRowPtr</code></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><code>bsrColInd</code></td>
<td>integer array of ( \text{nnzb} ) column indices of the non-zero blocks of matrix ( A ).</td>
</tr>
<tr>
<td><code>rowBlockDim</code></td>
<td>number of rows within a block of ( A ).</td>
</tr>
<tr>
<td><code>colBlockDim</code></td>
<td>number of columns within a block of ( A ).</td>
</tr>
<tr>
<td><code>copyValues</code></td>
<td>CUSPARSE_ACTION_SYMBOLIC or CUSPARSE_ACTION_NUMERIC.</td>
</tr>
<tr>
<td><code>baseIdx</code></td>
<td>CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td><code>pBufferSize</code></td>
<td>host pointer containing number of bytes of the buffer used in <code>gebsr2gebsc()</code></td>
</tr>
<tr>
<td><code>pBuffer</code></td>
<td>buffer allocated by the user; the size is return by <code>gebsr2gebsc_bufferSize()</code></td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Handle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bscVal</code></td>
<td>(&lt;\text{type}&gt;) array of ( \text{nnzb} \times \text{rowBlockDim} \times \text{colBlockDim} ) non-zero elements of matrix ( A ). It is only filled-in if <code>copyValues</code> is set to CUSPARSE_ACTION_NUMERIC.</td>
</tr>
<tr>
<td><code>bscRowInd</code></td>
<td>integer array of ( \text{nnzb} ) row indices of the non-zero blocks of matrix ( A ).</td>
</tr>
<tr>
<td><code>bscColPtr</code></td>
<td>integer array of ( \text{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](https://docs.nvidia.com/cusparse/index.html) for the description of the return status

### 13.3. `cusparse<t>gebsr2gebsr()`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusparseSgebsr2gebsr_bufferSize</td>
<td>Calculates the buffer size required for the conversion of csr to csc format.</td>
</tr>
<tr>
<td>cusparseDgebsr2gebsr_bufferSize</td>
<td>Calculates the buffer size required for the conversion of csr to csc format.</td>
</tr>
<tr>
<td>cusparseCgebsr2gebsr_bufferSize</td>
<td>Calculates the buffer size required for the conversion of csr to csc format.</td>
</tr>
<tr>
<td>cusparseZgebsr2gebsr_bufferSize</td>
<td>Calculates the buffer size required for the conversion of csr to csc format.</td>
</tr>
</tbody>
</table>
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)
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, 
int* bsrRowPtrC, 
int bsrColIndC, 
int rowBlockDimC, 
int colBlockDimC, 
void* pBuffer)
cusparseStatus_t
cusparseDgebsr2gebsr(cusparseHandle_t handle, 
cusparseDirection_t dir, 
int mb, 
int nb, 
int nnzb, 
const cusparseMatDescr_t descrA, 
const double* bsrValA, 
const int* bsrRowPtrA, 
const int* bsrColIndA, 
int rowBlockDimA, 
int colBlockDimA, 
const cusparseMatDescr_t descrC, 
double* bsrValC, 
int* bsrRowPtrC, 
int* bsrColIndC, 
int rowBlockDimC, 
int colBlockDimC, 
void* pBuffer)
This function converts a sparse matrix in general BSR format that is defined by the three arrays \texttt{bsrValA}, \texttt{bsrRowPtrA}, and \texttt{bsrColIndA} into a sparse matrix in another general BSR format that is defined by arrays \texttt{bsrValC}, \texttt{bsrRowPtrC}, and \texttt{bsrColIndC}.

If \texttt{rowBlockDimA}=1 and \texttt{colBlockDimA}=1, \texttt{cusparse}[S|D|C|Z]gebsr2gebsr() is the same as \texttt{cusparse}[S|D|C|Z]csr2gebsr().

If \texttt{rowBlockDimC}=1 and \texttt{colBlockDimC}=1, \texttt{cusparse}[S|D|C|Z]gebsr2gebsr() is the same as \texttt{cusparse}[S|D|C|Z]gebsr2csr().

\(A\) is an \(m\times n\) sparse matrix where \(m(=mb\times \text{rowBlockDim})\) is the number of rows of \(A\), and \(n(=nb\times \text{colBlockDim})\) is the number of columns of \(A\). The general BSR format of \(A\) contains \(nnzb(=\text{bsrRowPtrA}[mb] - \text{bsrRowPtrA}[0])\) nonzero blocks. The matrix \(C\) is also general BSR format with a different block size, \texttt{rowBlockDimC}\times \texttt{colBlockDimC}. If \(m\) is not a multiple of \texttt{rowBlockDimC}, or \(n\) is not a multiple of \texttt{colBlockDimC}, zeros are filled in. The number of block rows of \(C\) is \(mc(=(m+\text{rowBlockDimC}-1)/\text{rowBlockDimC})\). The number of block rows of \(C\) is \(nc(=(n+\text{colBlockDimC}-1)/\text{colBlockDimC})\). The number of nonzero blocks of \(C\) is \(nnzc\).

The implementation adopts a two-step approach to do the conversion. First, the user allocates \texttt{bsrRowPtrC} of \(mc+1\) elements and uses function \texttt{cusparseXgebsr2gebsrNnz()}. 

```c
 cusparseCgebsr2gebsr(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, 
                       const cusparseMatDescr_t descrC, 
                       cuComplex* bsrValC, 
                       int* bsrRowPtrC, 
                       int* bsrColIndC, 
                       int rowBlockDimC, 
                       int colBlockDimC, 
                       void* pBuffer)

cusparseStatus_t 

cusparseZgebsr2gebsr(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, 
                      const cusparseMatDescr_t descrC, 
                      cuDoubleComplex* bsrValC, 
                      int* bsrRowPtrC, 
                      int* bsrColIndC, 
                      int rowBlockDimC, 
                      int colBlockDimC, 
                      void* pBuffer)
```
to determine the number of nonzero block columns per block row of matrix C. Second, the user gathers \(\text{nnzc}\) [number of non-zero block columns of matrix C] from either (\(\text{nnzc}=\text{nnzTotalDevHostPtr}\)) or (\(\text{nnzc}=[\text{bsrRowPtrC}[\text{mc}]-\text{bsrRowPtrC}[0])\)) and allocates \(\text{bsrValC}\) of \(\text{nnzc}\cdot\text{rowBlockDimC}\cdot\text{colBlockDimC}\) elements and \(\text{bsrColIndC}\) of \(\text{nnzc}\) integers.

Finally the function \(\text{cusparse}\{S|D|C|Z\}\text{gebsr2gebsr}()\) is called to complete the conversion.

The user must call \(\text{gebsr2gebsr}_{\text{ bufferSize}}()\) to know the size of the buffer required by \(\text{gebsr2gebsr}()\), allocate the buffer, and pass the buffer pointer to \(\text{gebsr2gebsr}()\).

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 \(\text{pBuffer} \neq \text{NULL}\)
- 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.
nzb | 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()`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>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)</code></td>
<td></td>
</tr>
<tr>
<td><code>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)</code></td>
<td></td>
</tr>
<tr>
<td><code>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)</code></td>
<td></td>
</tr>
<tr>
<td><code>cusparseStatus_t cusparseZgebsr2csr(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)</code></td>
<td></td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status.
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></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 ). The supported matrix type is CUSPARSE_MATRIX_TYPE_GENERAL.</td>
</tr>
</tbody>
</table>
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()

cusparseStatus_t

cusparseScsr2gebsr_bufferSize(cusparseHandle_t handle,            
cusparseDirection_t dir,                                         
int m,                                                            
int n,                                                            
const cusparseMatDescr_t descrA,                                  
const float* csrValA,                                             
const int* csrRowPtrA,                                            
const int* csrColIndA,                                            
int rowBlockDim,                                                 
int colBlockDim,                                                 
int* pBufferSize)                                                 

cusparseStatus_t

cusparseDcsr2gebsr_bufferSize(cusparseHandle_t handle,            
cusparseDirection_t dir,                                         
int m,                                                            
int n,                                                            
const cusparseMatDescr_t descrA,                                  
int rowBlockDim,                                                 
int colBlockDim,                                                 
int* pBufferSize)                                                 


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,
rowBlockDim,
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,
rowBlockDim,
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,
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 $\text{csrValA}$, $\text{csrRowPtrA}$, and $\text{csrColIndA}$) into a sparse matrix $C$ in general BSR format (that is defined by the three arrays $\text{bsrValC}$, $\text{bsrRowPtrC}$, and $\text{bsrColIndC}$).
The matrix A is a \(m \times n\) sparse matrix and matrix C is a \((mb \times rowBlockDim) \times (nb \times colBlockDim)\) sparse matrix, where \(mb = \lfloor m/rowBlockDim \rfloor + 1\) is the number of block rows of C, and \(nb = \lfloor n/colBlockDim \rfloor + 1\) is the number of block columns of C.

The block of C is of size \(rowBlockDim \times colBlockDim\). If \(m\) is not a multiple of \(rowBlockDim\) or \(n\) is not a 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 \(\text{cusparseXcsr2gebsrNnz()}\) 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 \(\text{cusparse[S|D|C|Z]csr2gebsr()}\) is called to complete the conversion.

The user must obtain the size of the buffer required by \(\text{csr2gebsr()}\) by calling \(\text{csr2gebsr_bufferSize()}\), allocate the buffer, and pass the buffer pointer to \(\text{csr2gebsr()}\).

The general procedure is as follows:

```c
// Given CSR format (csrRowPtrA, csrColIndA, csrValA) and // blocks of BSR format are stored in column-major order.
cusparsesDirection_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,
   csrRowPtrA, csrColIndA,
   descrC,
   bsrValC, bsrRowPtrC, bsrColIndC,
   rowBlockDim, colBlockDim,
   pBuffer);
```

The routine \(\text{cusparseXcsr2gebsrNnz()}\) 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()

cusparseStatus_t
cusparseXcoo2csr(cusparseHandle_t handle, const int* cooRowInd, int nnz, int m, int* csrRowPtr, cusparseIndexBase_t idxBase)

This function converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).

It can also be used to convert the array containing the uncompressed column indices (corresponding to COO format) into an array of column pointers (corresponding to CSC format).

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

**Input**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<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**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

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>Input</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>Output</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()`

cusparseStatus_t cusparseXcsr2bsrNnz(cusparseHandle_t handle, cusparseDirection_t dir, int m, int n, const cusparseMatDescr_t descrA, const int* csrRowPtrA, const int* csrColIndA, int blockDim, const cusparseMatDescr_t descrC, int* bsrRowPtrC, int* nnzTotalDevHostPtr)

cusparseStatus_t cusparseScsr2bsr(cusparseHandle_t handle, cusparseDirection_t dir, int m, int n, const cusparseMatDescr_t descrA,
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 \( \text{nnzb} \) 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>(&lt;\text{type}&gt;) 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             | \(<\text{type}>\) 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()

```c

cusparseStatus_t
cusparseXcsr2coo(cusparseHandle_t handle, const int* csrRowPtr, int nnz,
                int m, int* cooRowInd,
                cusparseIndexBase_t idxBase)
```

This function converts the array containing the compressed row pointers (corresponding to CSR format) into an array of uncompressed row indices (corresponding to COO format).

It can also be used to convert the array containing the compressed column indices (corresponding to CSC format) into an array of uncompressed column indices (corresponding to COO format).
The routine requires no extra storage
The routine supports asynchronous execution
The routine supports CUDA graph capture

<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>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>

<table>
<thead>
<tr>
<th>Output</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()

```c
#include <cusparse.h>

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)
```

```c
#include <cusparse.h>

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,
                   cusparseCsr2CscAlg_t  alg,
                   size_t*              bufferSize)
```
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.

If `nnz == 0`, then `csrColInd`, `csrVal`, `cscVal`, and `cscRowInd` could have NULL value. In this case, `cscColPtr` is set to `idxBase` for all values.

If `m == 0` or `n == 0`, the pointers are not checked and the routine returns `CUSPARSE_STATUS_SUCCESS`.

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

**Input**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>cscVal</code></td>
<td>value array of size <code>nnz</code> of the CSC matrix; of same type as <code>valType</code></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cscColPtr</td>
<td>integer array of size $n + 1$ that contains the CSC column offsets</td>
</tr>
<tr>
<td>cscRowInd</td>
<td>integer array of size $nnz$ that contains the CSC row indices</td>
</tr>
<tr>
<td>valType</td>
<td>value type for both CSR and CSC matrices</td>
</tr>
<tr>
<td>copyValues</td>
<td>CUSPARSE_ACTION_SYMBOLIC or CUSPARSE_ACTION_NUMERIC</td>
</tr>
<tr>
<td>idxBase</td>
<td>Index base CUSPARSE_INDEX_BASE_ZERO or CUSPARSE_INDEX_BASE_ONE.</td>
</tr>
<tr>
<td>alg</td>
<td>Algorithm implementation. See cusparseCsr2CscAlg_t for possible values.</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 cuComplex* csrRowPtrA, const cuComplex* csrColIndA, cuComplex* A, int lda)
```
This function converts the sparse matrix in CSR format (that is defined by the three arrays \( \text{csrValA}, \text{csrRowPtrA}, \text{csrColIndA} \)) into the matrix \( \mathbf{A} \) in dense format. The dense matrix \( \mathbf{A} \) 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

**handle**
- handle to the cuSPARSE library context.

**m**
- number of rows of matrix \( \mathbf{A} \).

**n**
- number of columns of matrix \( \mathbf{A} \).

**descrA**
- the descriptor of matrix \( \mathbf{A} \). 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}.

**csrValA**
- \(<\text{type}>\) array of \( \text{nnz} = \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0) \) nonzero elements of matrix \( \mathbf{A} \).

**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 \( \mathbf{A} \).

**lda**
- leading dimension of array \( \mathbf{A} \).

### Output

**\( \mathbf{A} \)**
- array of dimensions \((\text{lda}, n)\) that is filled in with the values of the sparse matrix.

See \text{cusparseStatus_t} for the description of the return status.
13.12. cusparse<t>csr2csr_compress()

cusparseStatus_t
cusparseScsr2csr_compress(cusparseHandle_t handle, int m, int n, const cusparseMatDescr_t descrA, const float* csrValA, const int* csrColIndA, const int* csrRowPtrA, int nnzA, const int* nnzPerRow, float* csrValC, int* csrColIndC, int* csrRowPtrC, float tol)

cusparseStatus_t
cusparseDcsr2csr_compress(cusparseHandle_t handle, int m, int n, const cusparseMatDescr_t descrA, const double* csrValA, const int* csrColIndA, const int* csrRowPtrA, int nnzA, const int* nnzPerRow, double* csrValC, int* csrColIndC, int* csrRowPtrC, double tol)

cusparseStatus_t
cusparseCcsr2csr_compress(cusparseHandle_t handle, int m, int n, const cusparseMatDescr_t descrA, const cuComplex* csrValA, const int* csrColIndA, const int* csrRowPtrA, int nnzA, const int* nnzPerRow, cuComplex* csrValC, int* csrColIndC, int* csrRowPtrC, cuComplex tol)

cusparseStatus_t
cusparseZcsr2csr_compress(cusparseHandle_t handle, int m, int n, const cusparseMatDescr_t descrA, const cuDoubleComplex* csrValA, const int* csrColIndA, const int* csrRowPtrA, int nnzA, const int* nnzPerRow, cuDoubleComplex tol)
This function compresses the sparse matrix in CSR format into compressed CSR format. Given a sparse matrix \( A \) and a non-negative value threshold, 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 \( \text{cusparse<t>nnz_compress()} \) to determine \( \text{nnzPerRow} \) (the number of nonzeros columns per row) and \( \text{nnzC} \) (the total number of nonzeros). Second, the user allocates \( \text{csrValC} \) of \( \text{nnzC} \) elements and \( \text{csrColIndC} \) of \( \text{nnzC} \) integers. Finally, function \( \text{cusparse<t>csr2csr_compress()} \) 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>\textit{handle}</td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td>\textit{m}</td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td>\textit{n}</td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td>\textit{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>\textit{csrValA}</td>
<td>\text{&lt;type&gt;} array of ( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) elements of matrix ( A ).</td>
</tr>
<tr>
<td>\textit{csrColIndA}</td>
<td>integer array of ( \text{nnz}(= \text{csrRowPtrA}(m) - \text{csrRowPtrA}(0)) ) column indices of the elements of matrix ( A ).</td>
</tr>
<tr>
<td>\textit{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>\textit{nnzA}</td>
<td>number of nonzero elements in matrix ( A ).</td>
</tr>
<tr>
<td>\textit{nnzPerRow}</td>
<td>this array contains the number of elements kept in the compressed matrix, by row.</td>
</tr>
<tr>
<td>\textit{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>Function</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 \n",__func__,__FILE__,__LINE__); 
  exit(-1);}} while(0)
#define CUDA_CALL(X) ERR_NE((X),cudaSuccess)
#define CUSPARSE_CALL(X) ERR_NE((X),CUSPARSE_STATUS_SUCCESS)

int main(){
    int m = 6, n = 5;
    cusparseHandle_t  handle;
    CUSPARSE_CALL( cusparseCreate(&handle) );
    cusparseMatDescr_t descrX;
    CUSPARSE_CALL(cusparseCreateMatDescr(&descrX));

    // Initialize sparse matrix
    float *X;
    CUDA_CALL(cudaMallocManaged( &X, sizeof(float) * m * n ));
    memset( X, 0, sizeof(float) * m * n );
    X[0 + 0*m] = 1.0;  X[0 + 1*m] = 3.0;
    X[1 + 1*m] = -4.0; X[1 + 2*m] = 5.0;
    X[2 + 0*m] = 2.0;  X[2 + 3*m] = 7.0;  X[2 + 4*m] = 8.0;
    X[3 + 2*m] = 6.0;  X[3 + 4*m] = 9.0;
    X[4 + 3*m] = 3.5;  X[4 + 4*m] = 5.5;
    X[5 + 0*m] = 6.5;  X[5 + 2*m] = -9.9;

    // Initialize total_nnz, and  nnzPerRowX for cusparseSdense2csr()
    int total_nnz = 13;
    int *nnzPerRowX;
    CUDA_CALL( cudaMallocManaged( &nnzPerRowX, sizeof(int) * m ));
    nnzPerRowX[0] = 2;  nnzPerRowX[1] = 2;  nnzPerRowX[2] = 3;

    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,
                                       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
cusparseSdense2csc(cusparseHandle_t handle,
int m, int n,
const cusparseMatDescr_t descrA,
const float* A,
int lda,
const int* nnzPerCol,
float* cscValA,
int* cscRowIndA,
int* cscColPtrA)
cuSPARSE Format Conversion Reference

This function converts the matrix \( A \) in dense format into a sparse matrix in CSC format. All the parameters are assumed to have been pre-allocated by the user, and the arrays are filled in based on `nnzPerCol`, which can be precomputed with `cusparse<t>nnz()`.

- This function requires 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><code>handle</code></td>
<td>handle to the cuSPARSE library context.</td>
</tr>
<tr>
<td><code>m</code></td>
<td>number of rows of matrix ( A ).</td>
</tr>
<tr>
<td><code>n</code></td>
<td>number of columns of matrix ( A ).</td>
</tr>
<tr>
<td><code>descrA</code></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><code>A</code></td>
<td>array of dimensions (lda, n).</td>
</tr>
<tr>
<td>lda</td>
<td>leading dimension of dense array $A$.</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>nnzPerCol</td>
<td>array of size $n$ containing the number of nonzero elements per column.</td>
</tr>
</tbody>
</table>

### Output

- **cscValA**: <type> array of $\text{nnz}(= \text{cscRowPtrA}(m) - \text{cscRowPtrA}(0))$ nonzero elements of matrix $A$. It is only filled in if `copyValues` is set to `CUSPARSE_ACTION_NUMERIC`.
- **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 [cusparseStatus_t](#) for the description of the return status.

### 13.14. cusparse<t>dense2csr()

**[DEPRECATED]**

Use **cusparseDense2Sparse()** instead. *The routine will be removed in the next major release.*

```c
#include <cusparse.h>

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 float* A, int lda, const int* nnzPerRow, float* csrValA, int* csrRowPtrA, int* csrColIndA)
```

```c
#include <cusparse.h>

cusparseStatus_t cusparseCdense2csr(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)
```
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>lda</td>
<td>leading dimension of dense array ( A ).</td>
</tr>
<tr>
<td>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>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 column and the end of the last column plus one.</td>
</tr>
</tbody>
</table>
### 13.15. 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

---

csrColIndA | integer array of nnz (= csrRowPtrA(m) − csrRowPtrA(0)) column indices of the non-zero elements of matrix A.

See [cusparseStatus_t](#) for the description of the return status.
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>dirA</td>
<td></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></td>
<td>number of rows of matrix $A$.</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>number of columns of matrix $A$.</td>
</tr>
<tr>
<td>descrA</td>
<td></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>$A$</td>
<td></td>
<td>array of dimensions $(lda, n)$.</td>
</tr>
<tr>
<td>lda</td>
<td></td>
<td>leading dimension of dense array $A$.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnzPerRowColumn</td>
<td></td>
<td>array of size $m$ or $n$ containing the number of nonzero elements per row or column, respectively.</td>
</tr>
<tr>
<td>nnzTotalDevHostPtr</td>
<td></td>
<td>total number of nonzero elements in device or host memory.</td>
</tr>
</tbody>
</table>

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

**13.16. cusparseCreateIdentityPermutation()**

```c

cusparseStatus_t cusparseCreateIdentityPermutation(cusparseHandle_t handle,
                                                  int              n,
                                                  int*             p);
```

This function creates an identity map. The output parameter $p$ represents such map by $p = 0:1:(n-1)$.

This function is typically used with coosort, csrsort, cscsort.

- 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>
</tbody>
</table>
\[ n \quad \text{host} \quad \text{size of the map.} \]

**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 [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 $\text{cooVal}$, $P$ must be set as $0:1:([\text{nnz}-1])$ before $\text{coosort()}$, and after $\text{coosort()}$, new sorted value array satisfies $\text{cooVal\_sorted} = \text{cooVal}(P)$.

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 $\text{coosort()}$.

- The routine requires no extra storage if $\text{ pBuffer } \neq \text{ NULL}$
- 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>$\text{nnz}$</td>
<td>host</td>
<td>number of nonzero elements of matrix $A$.</td>
</tr>
<tr>
<td>$\text{cooRows}$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ unsorted row indices of $A$.</td>
</tr>
<tr>
<td>$\text{cooCols}$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ unsorted column indices of $A$.</td>
</tr>
<tr>
<td>$P$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ unsorted map indices. To construct $\text{cooVal}$, the user has to set $P=0:1:([\text{nnz}-1])$.</td>
</tr>
<tr>
<td>$\text{ pBuffer}$</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by $\text{coosort_bufferSizeExt()}$.</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>parameter</th>
<th>device or host</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cooRows}$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ sorted row indices of $A$.</td>
</tr>
<tr>
<td>$\text{cooCols}$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ sorted column indices of $A$.</td>
</tr>
<tr>
<td>$P$</td>
<td>device</td>
<td>integer array of $\text{nnz}$ sorted map indices.</td>
</tr>
<tr>
<td>$\text{ pBuffer_size_in_bytes}$</td>
<td>host</td>
<td>number of bytes of the buffer.</td>
</tr>
</tbody>
</table>

See $\text{cusparseStatus\_t}$ for the description of the return status

### 13.18. $\text{cusparseXcsrsort()}$

```c

* $\text{cusparseXcsrsort\_bufferSizeExt}(\text{cusparseHandle\_t handle, int m, int n, int nnz, const int* csrRowPtr, const int* csrColInd,}$

```
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 |
// A = | 4 5 6 |
//    | 7 8 9 |
const int m = 3;
const int n = 3;
const int nnz = 9;
csrRowPtr[m+1] = { 0, 3, 6, 9}; // on device
csrColInd[nnz] = { 2, 1, 0, 0, 2,1, 1, 2, 0}; // on device
csrVal[nnz] = { 3, 2, 1, 4, 6, 5, 8, 9, 7}; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
 cusparseXcsrsort_bufferSizeExt(handle, m, n, nnz, csrRowPtr, csrColInd, &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSR format
 cusparseXcsrsort(handle, m, n, nnz, descrA, csrRowPtr, csrColInd, P, pBuffer);

// step 4: gather sorted csrVal
 cusparseDgthr(handle, nnz, csrVal, csrVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

- 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>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>P</td>
<td>device</td>
<td>integer array of $nnz$ unsorted map indices. To construct $csrVal$, 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 <code>csrsort_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>csrColInd</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

13.19. `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, const 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:

```c
// 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 <code>n+1</code> 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 <code>nnz</code> unsorted row indices of A.</td>
</tr>
<tr>
<td>P</td>
<td>device</td>
<td>integer array of <code>nnz</code> unsorted map indices. To construct <code>cscVal</code>, the user has to set <code>P=0:1:nnz-1</code>.</td>
</tr>
</tbody>
</table>
pBuffer device buffer allocated by the user; the size is returned by
cscsort_bufferSizeExt()

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,
cusparseStatus_t
cusparseScsr2csru(cusparseHandle_t handle,
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
float* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csruInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseDcsr2csru(cusparseHandle_t handle,
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
double* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csruInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseCcsr2csru(cusparseHandle_t handle,
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
cuComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csruInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseZcsr2csru(cusparseHandle_t handle,
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
cuDoublingComplex* csrVal,
const int* csrRowPtr,
int* csrColInd,
csr2csruInfo_t info,
void* pBuffer)
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.

```plaintext
do
    step 1: compute residual vector $r = b - A x^{(k)}$
    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 $z = L^{(-1)} \times (b)$
    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 `cudaMalloc` inside the conversion routine. Instead, the user has to provide the buffer explicitly.
4. The conversion between unsorted CSR and sorted CSR may needs several times, but the function only generates the permutation vector $P$ once.
5. The function is based on `csrsort`, `gather` and `scatter` operations.

The operation is called `csru2csr`, which means unsorted CSR to sorted CSR. Also we provide the inverse operation, called `csr2csru`.

In order to keep the permutation vector invisible, we need an opaque structure called `csru2csrInfo`. Then two functions (`cusparseCreateCsru2csrInfo`, `cusparseDestroyCsru2csrInfo`) are used to initialize and to destroy the opaque structure.

`cusparse[S|D|C|Z]csru2csr_bufferSizeExt` returns the size of the buffer. The permutation vector $P$ is also allocated inside `csru2csrInfo`. The lifetime of the permutation vector is the same as the lifetime of `csru2csrInfo`.

`cusparse[S|D|C|Z]csru2csr` performs forward transformation from unsorted CSR to sorted CSR. First call uses `csrsort` to generate the permutation vector $P$, and subsequent call uses $P$ to do transformation.

`cusparse[S|D|C|Z]csr2csru` performs backward transformation from sorted CSR to unsorted CSR. $P$ is used to get unsorted form back.

The routine `cusparse<t>csru2csr()` has the following properties:

- The routine requires no extra storage if pBuffer != NULL
- The routine does not support asynchronous execution
- The routine does not support CUDA graph capture

The routine `cusparse<t>csr2csru()` has the following properties if pBuffer != 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 CUSPARSE_MATRIX_TYPE_GENERAL. Also, the supported index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.</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 cusparseCreateCsru2csrInfo().</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by csru2csr_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>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>pBuffer</td>
<td>device</td>
<td>integer array of nnz sorted column indices of A.</td>
</tr>
<tr>
<td>pBuffer</td>
<td>device</td>
<td>integer array of nnz sorted column indices of A.</td>
</tr>
</tbody>
</table>

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

### 13.21. cusparseXpruneDense2csr()

```cpp
cusparseStatus_t cusparseXpruneDense2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
const __half* A,
int lda,
const __half* threshold,
const cusparseMatDescr_t descrC,
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusparseSpruneDense2csr_bufferSizeExt</td>
<td>Calculate buffer size for CSR conversion</td>
<td>cusparseHandle_t handle, int m, int n, float* A, int lda, float* threshold, cusparseMatDescr_t descrC, float* csrValC, int* csrRowPtrC, int* csrColIndC, size_t* pBufferSizeInBytes</td>
</tr>
<tr>
<td>cusparseDpruneDense2csr_bufferSizeExt</td>
<td></td>
<td>cusparseHandle_t handle, int m, int n, double* A, int lda, double* threshold, cusparseMatDescr_t descrC, double* csrValC, int* csrRowPtrC, int* csrColIndC, size_t* pBufferSizeInBytes</td>
</tr>
<tr>
<td>cusparseHpruneDense2csrNnz</td>
<td>Calculate number of non-zero elements</td>
<td>cusparseHandle_t handle, int m, int n, __half* A, int lda, __half* threshold, cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer</td>
</tr>
<tr>
<td>cusparseSpruneDense2csrNnz</td>
<td></td>
<td>cusparseHandle_t handle, int m, int n, float* A, int lda, float* threshold, cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer</td>
</tr>
<tr>
<td>cusparseDpruneDense2csrNnz</td>
<td></td>
<td>cusparseHandle_t handle, int m, int n, double* A, int lda, double* threshold, cusparseMatDescr_t descrC, int* csrRowPtrC, int* nnzTotalDevHostPtr, void* pBuffer</td>
</tr>
</tbody>
</table>

**Parameters:**
- `handle` (cusparseHandle_t): A handle to the cuSPARSE context.
- `m` (int): Number of rows of the matrix.
- `n` (int): Number of columns of the matrix.
- `A` (float* or double*): Pointer to the dense matrix.
- `lda` (int): Leading dimension of the matrix.
- `threshold` (float or double): Pruning threshold.
- `descrC` (cusparseMatDescr_t): Description of the CSR matrix.
- `csrValC` (float* or double*): Pointer to the values in CSR.
- `csrRowPtrC` (int*): Pointer to the row pointers in CSR.
- `csrColIndC` (int*): Pointer to the column indices in CSR.
- `pBufferSizeInBytes` (size_t*): Pointer to the buffer size in bytes.
- `nnzTotalDevHostPtr` (int*): Pointer to the total number of non-zero elements in CSR.
- `pBuffer` (void*): Pointer to the buffer for CSR.
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 pruneDense2csrNnz() to determine the number of nonzeros columns per row. Second, the user gathers $\text{nnzC}$ [number of nonzeros of matrix C] from either ($\text{nnzC} = \ast \text{nnzTotalDevHostPtr}$) or ($\text{nnzC} = \text{csrRowPtrC}[m]$)
csrRowPtrC[0]) and allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function pruneDense2csr() is called to complete the conversion.

The user must obtain the size of the buffer required by pruneDense2csr() by calling pruneDense2csr_bufferSizeExt(), allocate the buffer, and pass the buffer pointer to pruneDense2csr().

Appendix 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 [lda, 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 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 pruneDense2csr_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 host</td>
<td>number of bytes of the buffer.</td>
<td></td>
</tr>
</tbody>
</table>

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

### 13.22. cusparseXpruneCsr2csr()

cusparseStatus_t
cusparseXpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const _half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const _half* threshold,
const cusparseMatDescr_t descrC,
const _half* csrValC,
const int* csrRowPtrC,
const int*csrColIndC,
size_t*
pBufferSizeInBytes)

cusparseStatus_t
cusparseXpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const float* threshold,
const cusparseMatDescr_t descrC,
const float* csrValC,
const int* csrRowPtrC,
const int*csrColIndC,
size_t*
pBufferSizeInBytes)

cusparseStatus_t
cusparseXpruneCsr2csr_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnzA,
const cusparseMatDescr_t descrA,
const double* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const double* threshold,
const cusparseMatDescr_t descrC,
const double* csrValC,
const int* csrRowPtrC,
const int*csrColIndC,
size_t*
pBufferSizeInBytes)
cusparseStatus_t
cusparseHpruneCsr2csrNnz(cusparseHandle_t handle, int m, int n, int nnzA,
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const __half* threshold,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* csrColIndC,
__half* csrValC,
int* csrRowPtrC,
int* csrColIndC,
void* pBuffer)

cusparseStatus_t
cusparseSpruneCsr2csrNnz(cusparseHandle_t handle, int m, int n, int nnzA,
const cusparseMatDescr_t descrA,
const float* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const float* threshold,
const cusparseMatDescr_t descrC,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseDpruneCsr2csrNnz(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,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
void* pBuffer)

cusparseStatus_t
cusparseHpruneCsr2csr(cusparseHandle_t handle, int m, int n, int nnzA,
const cusparseMatDescr_t descrA,
const __half* csrValA,
const int* csrRowPtrA,
const int* csrColIndA,
const __half* threshold,
const cusparseMatDescr_t descrC,
const __half* csrValC,
const int* csrRowPtrC,
const int* csrColIndC,
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

$$C(i,j) = \begin{cases} A(i,j) & \text{if } |A(i,j)| > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates $\text{csrRowPtrC}$ of $m+1$ elements and uses function $\text{pruneCsr2csrNnz()}$ 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 $\text{pruneCsr2csr()}$ is called to complete the conversion.

The user must obtain the size of the buffer required by $\text{pruneCsr2csr()}$ by calling $\text{pruneCsr2csr_bufferSizeExt()}$, allocate the buffer, and pass the buffer pointer to $\text{pruneCsr2csr()}$.

Appendix section provides a simple example of $\text{pruneCsr2csr()}$.

The routine $\text{cusparse<t>pruneCsr2csrNnz()}$ 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 ( \text{CUSPARSE_MATRIX_TYPE_GENERAL} ), Also, the supported index bases are ( \text{CUSPARSE_INDEX_BASE_ZERO} ) and ( \text{CUSPARSE_INDEX_BASE_ONE} ).</td>
</tr>
<tr>
<td>csrValA</td>
<td>device</td>
<td>&lt;type&gt; array of ( \text{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 ( \text{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 ( \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>pBuffer</td>
<td>device</td>
<td>buffer allocated by the user; the size is returned by ( \text{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 ( \text{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 ( \text{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

cusparseXpruneDense2csrPercentage_bufferSizeExt(cusparseHandle_t
handle,

int

m,

int

n,

const __half*

A,

int

lda,

float

percentage,

cnstrMatDescr_t
descrC,

const __half*
csrValC,

const int*
csrRowPtrC,

const int*
csrColIndC,

pruneInfo_t

info,

size_t*
pBufferSizeInBytes)

cusparseStatus_t

cusparseSpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t
handle,

int

m,

int

n,

const float*

A,

int

lda,

float

percentage,

cnstrMatDescr_t
descrC,

const float*
csrValC,

const int*
csrRowPtrC,

const int*
csrColIndC,

pruneInfo_t

info,

size_t*
pBufferSizeInBytes)

cusparseStatus_t

cusparseDpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t
handle,
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>cusparseHpruneDense2csrNnzByPercentage</td>
<td>handle, m, n, A, lda, percentage, descrC, csrRowPtrC, nnzTotalDevHostPtr, pruneInfo_t, pBuffer</td>
</tr>
<tr>
<td>cusparseSpruneDense2csrNnzByPercentage</td>
<td>handle, m, n, A, lda, percentage, descrC, csrRowPtrC, nnzTotalDevHostPtr, pruneInfo_t, pBuffer</td>
</tr>
<tr>
<td>cusparseDpruneDense2csrNnzByPercentage</td>
<td>handle, m, n, A, lda, percentage, descrC, csrRowPtrC, nnzTotalDevHostPtr, pruneInfo_t, pBuffer</td>
</tr>
</tbody>
</table>
This function prunes a dense matrix to a sparse matrix by percentage.

Given a dense matrix $A$ and a non-negative value `percentage`, the function computes sparse matrix $C$ by the following three steps:

Step 1: sort absolute value of $A$ in ascending order.

$$\text{key} := \text{sort}(|A|)$$

Step 2: choose threshold by the parameter `percentage`
Step 3: call `pruneDense2csr()` by with the parameter `threshold`.

The implementation adopts a two-step approach to do the conversion. First, the user allocates `csrRowPtrC` of \( m+1 \) elements and uses function `pruneDense2csrNnzByPercentage()` to determine the number of nonzeros columns per row. Second, the user gathers `nnzC` [number of nonzeros of matrix `C`] from either \( \text{nnzC} = \ast_{\text{nnzTotalDevHostPtr}} \) or \( \text{nnzC} = \text{csrRowPtrC}[m] - \text{csrRowPtrC}[0] \) 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 <code>A</code>.</td>
</tr>
<tr>
<td>n</td>
<td>host</td>
<td>number of columns of matrix <code>A</code>.</td>
</tr>
<tr>
<td>A</td>
<td>device</td>
<td>array of dimension ([\text{lda}, n]).</td>
</tr>
<tr>
<td>lda</td>
<td>device</td>
<td>leading dimension of <code>A</code>. It must be at least ([1, m]).</td>
</tr>
<tr>
<td>percentage</td>
<td>host</td>
<td>percentage ([\leq 100 \text{ and } \geq 0])</td>
</tr>
<tr>
<td>descrC</td>
<td>host</td>
<td>the descriptor of matrix <code>C</code>. The supported matrix type is <code>CUSPARSE_MATRIX_TYPE_GENERAL</code>. Also, the supported</td>
</tr>
</tbody>
</table>
index bases are CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.

| pBuffer | device | buffer allocated by the user; the size is returned by pruneDense2csrByPercentage_bufferSizeExt(). |

Output

<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

13.24. cusparseXpruneCsr2csrByPercentage()

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)
cusparseStatus_t
cusparseSpruneCsr2csrByPercentage_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnzA,
descrA,
csrValA,
csrRowPtrA,
csrColIndA,
percentage,
descrC,
csrValC,
csrRowPtrC,
csrColIndC,
info,
pBufferSizeInBytes)
cusparseStatus_t
cusparseSpruneCsr2csrByPercentage_bufferSizeExt(cusparseHandle_t handle,
int m,
int n,
int nnzA,
descrA,
csrValA,
csrRowPtrA,
csrColIndA,
percentage,
descrC,
csrValC,
csrRowPtrC,
csrColIndC,
info,
pBufferSizeInBytes)
cusparseStatus_t
cusparseHpruneCsr2csrNnzByPercentage(cusparseHandle_t handle,
cusparseStatus_t
cusparseSpruneCsr2csrNnzByPercentage(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,
int* csrRowPtrC,
int* nnzTotalDevHostPtr,
pruneInfo_t info,
void* pBuffer)

cusparseStatus_t
cusparseDpruneCsr2csrNnzByPercentage(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)

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,
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 $\text{pruneCsr2csr()}$ 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 $\text{pruneCsr2csrNnzByPercentage()}$ to determine the number of nonzeros columns per row. Second, the user gathers
nnzC [number of nonzeros of matrix C] from either (nnzC=*nnzTotalDevHostPtr) or (nnzC=csrRowPtrC[m]−csrRowPtrC[0]) and allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function pruneCsr2csrByPercentage() is called to complete the conversion.

The user must obtain the size of the buffer required by pruneCsr2csrByPercentage() by calling pruneCsr2csrByPercentage_bufferSizeExt(), allocate the buffer, and pass the buffer pointer to pruneCsr2csrByPercentage().

Remark 1: the value of percentage must be not greater than 100. Otherwise, CUSPARSE_STATUS_INVALID_VALUE is returned.

Appendix 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 isCUSPARSE_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 isCUSPARSE_MATRIX_TYPEGENERAL. 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>
<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

13.25. cusparse<t>nnz_compress()

```
cusparseStatus_t cusparseSnnz_compress(cusparseHandle_t handle, int m, const cusparseMatDescr_t descr, const float* csrValA, const int* csrRowPtrA, int* nnzPerRow, int* nnzC, float tol)
cusparseStatus_t cusparseDnnz_compress(cusparseHandle_t handle, int m, const cusparseMatDescr_t descr, const double* csrValA, const int* csrRowPtrA, int* nnzPerRow, int* nnzC, double tol)
cusparseStatus_t cusparseCnnz_compress(cusparseHandle_t handle, int m, const cusparseMatDescr_t descr, const cuComplex* csrValA, const int* csrRowPtrA, int* nnzPerRow, int* nnzC, cuComplex tol)
cusparseStatus_t cusparseZnnz_compress(cusparseHandle_t handle, int m, const cusparseMatDescr_t descr, const cuDoubleComplex* csrValA, const int* csrRowPtrA, int* nnzPerRow, int* nnzC, cuDoubleComplex tol)
```
cuDoubleComplex tol)

This function is the step one to convert from csr format to compressed csr format.

Given a sparse matrix $A$ and a non-negative value threshold, the function returns
nnzPerRow (the number of nonzeros columns per row) and nnzC (the total number of nonzeros)
of a sparse matrix $C$, defined by

$$C(i,j) = A(i,j) \text{ if } |A(i,j)| > \text{threshold}$$

A key assumption for the cuComplex and cuDoubleComplex case is that this tolerance is given
as the real part. For example tol = 1e-8 + 0*i and we extract cureal, that is the x component of
this struct.

- 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.
Chapter 14. cuSPARSE Generic API Reference

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>
### 14.1.1. Value and Meaning of Data Types

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Data Type</th>
<th>Header</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)


### 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>
<tr>
<td>CUSPARSE_FORMAT_CSC</td>
<td>The matrix is stored in Compressed Sparse Column (CSC) format</td>
</tr>
<tr>
<td>CUSPARSE_FORMAT_BLOCKED_ELL</td>
<td>The matrix is stored in Blocked-Ellpack (Blocked-ELL) format</td>
</tr>
</tbody>
</table>

### 14.1.3. cusparseOrder_t

This type indicates the memory layout of a dense matrix.

<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.

<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()

```
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>HOST</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 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()

```
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/out</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>

See cusparseStatus_t for the description of the return status

### 14.2.3. cusparseSpVecGet()

```c
#define cusparseSpVecGet(t, spVecDescr, size, nnz, indices, values, idxType, idxBase, valueType)
```

This function returns the fields 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>size</td>
<td>HOST</td>
<td>OUT</td>
<td>Size of the sparse vector</td>
</tr>
<tr>
<td>nnz</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of non-zero entries of the sparse vector</td>
</tr>
<tr>
<td>indices</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Indices of the sparse vector. Array of size nnz</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse vector. Array of size nnz</td>
</tr>
<tr>
<td>idxType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the data type of indices</td>
</tr>
<tr>
<td>idxBase</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of indices</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

### 14.2.4. cusparseSpVecGetIndexBase()

```c
#define cusparseSpVecGetIndexBase(t, spVecDescr, idxBase)
```

This function returns the idxBase 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>idxBase</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the base index of indices</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status
14.2.5. cusparseSpVecGetValues()

cusparseStatus_t
cusparseSpVecGetValues(cusparseSpVecDescr_t spVecDescr, 
                         void**               values)

This function returns 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>OUT</td>
<td>Values of the sparse vector. Array of size (\text{nnz})</td>
</tr>
</tbody>
</table>

See cusparseStatus_t for the description of the return status.

14.2.6. cusparseSpVecSetValues()

cusparseStatus_t
cusparseSpVecSetValues(cusparseSpVecDescr_t spVecDescr, 
                       void*                values)

This function set 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 (\text{nnz})</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()

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).
### 14.3.2. cusparseCreateCooAoS() [DEPRECATED]

This function initializes the sparse matrix descriptor `spMatDescr` in the 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><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>cooInd</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>&lt;Row, Column&gt; indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>cooValues</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>cooIdxType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Data type of <code>cooInd</code> and <code>cooColInd</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>HOST</td>
<td>IN</td>
<td>Base index of <code>cooInd</code> and <code>cooColInd</code></td>
</tr>
<tr>
<td><code>valueType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Datatype of <code>cooValues</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.
14.3.3. cusparseCreateCsr()

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. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>csrRowOffsetsType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Data type of <code>csrRowOffsets</code></td>
</tr>
<tr>
<td><code>csrColIndType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Data type of <code>csrColInd</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>HOST</td>
<td>IN</td>
<td>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>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()
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>
<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>cscColOffsets</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Column offsets of the sparse matrix. Array of size <code>cols</code> + 1</td>
</tr>
<tr>
<td><code>cscRowInd</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Row indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>cscValues</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Values of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>cscColOffsetsType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Data type of <code>cscColOffsets</code></td>
</tr>
<tr>
<td><code>cscRowIndType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Data type of <code>cscRowInd</code></td>
</tr>
<tr>
<td><code>idxBase</code></td>
<td>HOST</td>
<td>IN</td>
<td>Base index of <code>cscColOffsets</code> and <code>cscRowInd</code></td>
</tr>
<tr>
<td><code>valueType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Datatype of <code>cscValues</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.

### 14.3.5. `cusparseCreateBlockedEll()`

This function initializes the sparse matrix descriptor `spMatDescr` for the Blocked-Ellpack (ELL) 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>ellBlockSize</code></td>
<td>HOST</td>
<td>IN</td>
<td>Size of the ELL-Block</td>
</tr>
</tbody>
</table>
### Param. | Memory | In/out | Meaning
---|---|---|---
`ellCols` | HOST | IN | Actual number of columns of the Blocked-Ellpack format (ellValue columns)
`ellColInd` | DEVICE | IN | Blocked-ELL Column indices. Array of size [ellCols / ellBlockSize][rows / ellBlockSize]
`ellValue` | DEVICE | IN | Values of the sparse matrix. Array of size rows * ellCols
`ellIdxType` | HOST | IN | Data type of ellColInd
`idxBase` | HOST | IN | Base index of ellColInd
`valueType` | HOST | IN | Datatype of ellValue

Blocked-ELL Column indices (ellColInd) are in the range \([0, cols / ellBlockSize - 1]\). The array can contain \(-1\) values for indicating empty blocks.

**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).

#### Figure 1. Blocked-ELL representation

![Blocked-ELL representation](image)

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

### 14.3.6. cusparseDestroySpMat()

```c
 cusparseStatus_t
cusparseDestroySpMat(cusparseSpMatDescr_t spMatDescr)
```

This function releases the host memory allocated for the sparse matrix descriptor `spMatDescr`. 

---

**cuSPARSE Library** 

DU-06709-001_v11.4 | 245
**Param.** | **Memory** | **In/out** | **Meaning** |
--- | --- | --- | --- |
spMatDescr | HOST | IN | Sparse matrix descriptor |
rows | HOST | OUT | Number of rows of the sparse matrix |
cols | HOST | OUT | Number of columns of the sparse matrix |
nnz | HOST | OUT | Number of non-zero entries of the sparse matrix |
cooRowInd | DEVICE | OUT | Row indices of the sparse matrix. Array of size nnz |
cooColInd | DEVICE | OUT | Column indices of the sparse matrix. Array of size nnz |
cooValues | DEVICE | OUT | Values of the sparse matrix. Array of size nnz |
cooIdxType | HOST | OUT | Data type of cooRowInd and cooColInd |
idxBase | HOST | OUT | Base index of cooRowInd and cooColInd |
valueType | HOST | OUT | Datatype of cooValues |

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

### 14.3.7. cusparseCooGet()

```c
cusparseStatus_t
cusparseCooGet(cusparseSpMatDescr_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].

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

### 14.3.8. cusparseCooAosGet() [DEPRECATED]

```c
cusparseStatus_t
cusparseCooAoSGet(cusparseSpMatDescr_t spMatDescr,
       int64_t* rows,
       int64_t* cols,
       int64_t* nnz,
       void** cooInd,
       void** cooValues,
       cusparseIndexType_t* idxType,
       cusparseIndexBase_t* idxBase,
```

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

cusparseStatus_t  
cusparseCsrGet(cusparseSpMatDescr_t spMatDescr,  
int64_t* rows,  
int64_t* cols,  
int64_t* nnz,  
void** csrRowOffsets,  
void** csrColInd,  
void** csrValues,  
cusparseIndexType_t* csrRowOffsetsType,  
cusparseIndexType_t* csrColIndType,  
cusparseIndexBase_t* idxBase,  
cudaDataType* valueType)

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>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>csrRowOffsets</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Row offsets of the sparse matrix. Array of size rows + 1</td>
</tr>
<tr>
<td>csrColInd</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Column indices of the sparse matrix</td>
</tr>
<tr>
<td>csrValues</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the sparse matrix. Array of size nnz</td>
</tr>
<tr>
<td>csrRowOffsetsType</td>
<td>HOST</td>
<td>OUT</td>
<td>Data type of csrRowOffsets</td>
</tr>
<tr>
<td>csrColIndType</td>
<td>HOST</td>
<td>OUT</td>
<td>Data type of csrColInd</td>
</tr>
</tbody>
</table>
14.3.10. cusparseCsrSetPointers()

cusparseStatus_t
cusparseCsrSetPointers(cusparseSpMatDescr_t spMatDescr,
    void*                csrRowOffsets,
    void*                csrColInd,
    void*                csrValues)

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>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>csrRowOffsets</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>csrColInd</td>
<td>DEVICE</td>
<td>IN</td>
<td>Column indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td>csrValues</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.

14.3.11. cusparseCscSetPointers()

cusparseStatus_t
cusparseCscSetPointers(cusparseSpMatDescr_t spMatDescr,
    void*                cscColOffsets,
    void*                cscRowInd,
    void*                cscValues)

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>spMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>cscColOffsets</td>
<td>DEVICE</td>
<td>IN</td>
<td>Col offsets of the sparse matrix. Array of size <code>cols + 1</code></td>
</tr>
<tr>
<td>cscRowInd</td>
<td>DEVICE</td>
<td>IN</td>
<td>Row indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td>cscValues</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.

14.3.12. cusparseCooSetPointers()
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>cooRows</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Row indices of the sparse matrix. Array of size <code>nnz</code></td>
</tr>
<tr>
<td><code>cooColumns</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>cooValues</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.

### 14.3.13. `cusparseBlockedEllGet()`

This function returns the fields of the sparse matrix descriptor `spMatDescr` stored in Blocked-Ellpack (ELL) 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>ellBlockSize</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Size of the ELL-Block</td>
</tr>
<tr>
<td><code>ellCols</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Actual number of columns of the Blocked-Ellpack format</td>
</tr>
</tbody>
</table>
| `ellColInd` | DEVICE     | OUT    | Column indices for the ELL-Block. Array of size `cols / ellBlockSize`[
| `ellValue` | DEVICE     | OUT    | Values of the sparse matrix. Array of size `rows * ellCols`            |
| `ellIdxType` | HOST     | OUT    | Data type of `ellColInd`                                               |
| `idxBase`  | HOST       | OUT    | Base index of `ellColInd`                                              |
| `valueType` | HOST       | OUT    | Datatype of `ellValue`                                                 |

See `cusparseStatus_t` for the description of the return status.
14.3.14. cusparseSpMatGetSize()

cusparseStatus_t
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 \texttt{cusparseStatus_t} for the description of the return status.

14.3.15. cusparseSpMatGetFormat()

cusparseStatus_t
cusparseSpMatGetFormat(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>Storage format of the sparse matrix</td>
</tr>
</tbody>
</table>

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

14.3.16. cusparseSpMatGetIndexBase()

cusparseStatus_t
cusparseSpMatGetIndexBase(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>Base index of the sparse matrix</td>
</tr>
</tbody>
</table>

See \texttt{cusparseStatus_t} for the description of the return status.
14.3.17. cusparseSpMatGetValues()

```c
 cusparseStatus_t
 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><code>spMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>values</code></td>
<td>DEVICE</td>
<td>OUT</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.

14.3.18. cusparseSpMatSetValues()

```c
 cusparseStatus_t
 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><code>spMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>values</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.

14.3.19. cusparseSpMatGetStridedBatch()

```c
 cusparseStatus_t
 cusparseSpMatGetStridedBatch(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><code>spMatDescr</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td><code>batchCount</code></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.
14.3.20. cusparseSpMatSetStridedBatch()

[[DEPRECATED]] use cusparseSpMatSetCsrStridedBatch(), cusparseSpMatSetCooStridedBatch() instead. The routine will be removed in the next major release.

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.

14.3.21. cusparseCooSetStridedBatch()

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.

14.3.22. cusparseCsrSetStridedBatch()

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.
### 14.3.23. `cusparseSpMatGetAttribute()`

```c
 cusparseStatus_t
cusparseSpMatGetAttribute(cusparseSpMatDescr_t spMatDescr,
                          cusparseSpMatAttribute_t attribute,
                          void* data,
                          size_t dataSize)
```

The function gets the attributes 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>attribute</code></td>
<td>HOST</td>
<td>IN</td>
<td>Attribute enumerator</td>
</tr>
<tr>
<td><code>data</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Attribute value</td>
</tr>
<tr>
<td><code>dataSize</code></td>
<td>HOST</td>
<td>IN</td>
<td>Size of the attribute in bytes for safety</td>
</tr>
</tbody>
</table>

- **Attribute**
  - **CUSPARSE_SPMAT_FILL_MODE**: Indicates if the lower or upper part of a matrix is stored in sparse storage.
    - **Possible Values**: `CUSPARSE_FILL_MODE_LOWER` and `CUSPARSE_FILL_MODE_UPPER`.
  - **CUSPARSE_SPMAT_DIAG_TYPE**: Indicates if the matrix diagonal entries are unity.
    - **Possible Values**: `CUSPARSE_DIAG_TYPE_NON_UNIT` and `CUSPARSE_DIAG_TYPE_UNIT`.

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

### 14.3.24. `cusparseSpMatSetAttribute()`

```c
 cusparseStatus_t
cusparseSpMatSetAttribute(cusparseSpMatDescr_t spMatDescr,
                          cusparseSpMatAttribute_t attribute,
                          const void* data,
                          size_t dataSize)
```

The function sets the attributes 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>attribute</code></td>
<td>HOST</td>
<td>IN</td>
<td>Attribute enumerator</td>
</tr>
<tr>
<td><code>data</code></td>
<td>HOST</td>
<td>IN</td>
<td>Attribute value</td>
</tr>
<tr>
<td><code>dataSize</code></td>
<td>HOST</td>
<td>IN</td>
<td>Size of the attribute in bytes for safety</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>spMatDescr</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse matrix descriptor</td>
</tr>
<tr>
<td>attribute</td>
<td>HOST</td>
<td>IN</td>
<td>Attribute enumerator</td>
</tr>
<tr>
<td>data</td>
<td>HOST</td>
<td>IN</td>
<td>Attribute value</td>
</tr>
<tr>
<td>dataSize</td>
<td>HOST</td>
<td>IN</td>
<td>Size of the attribute in bytes for safety</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SPMAT_FILL_MODE</td>
<td>Indicates if the lower or upper part of a matrix is stored in sparse storage</td>
<td>CUSPARSE_FILL_MODE_LOWER, CUSPARSE_FILL_MODE_UPPER</td>
</tr>
<tr>
<td>CUSPARSE_SPMAT_DIAG_TYPE</td>
<td>Indicates if the matrix diagonal entries are unity</td>
<td>CUSPARSE_DIAG_TYPE_NON_UNIT, CUSPARSE_DIAG_TYPE_UNIT</td>
</tr>
</tbody>
</table>

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

### 14.4. Dense Vector APIs

The cuSPARSE helper functions for dense vector descriptor are described in this section.

#### 14.4.1. cusparseCreateDnVec()

```
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()

```
cusparseStatus_t
```

---

See [cusparseStatus_t](#) for the description of the return status
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/out</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(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(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>
</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 size</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the datatype of values</td>
</tr>
<tr>
<td>order</td>
<td>HOST</td>
<td>IN</td>
<td>Enumerator specifying the memory layout of the dense matrix</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</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>
14.5.3. cusparseDnMatGet()

```
cusparseStatus_t
cusparseDnMatGet(cusparseDnMatDescr_t dnMatDescr,
                 int64_t*             rows,
                 int64_t*             cols,
                 int64_t*             ld,
                 void**               values,
                 cudaDataType*        type,
                 cusparseOrder_t*     order)
```

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>dnMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td>rows</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of rows of the dense matrix</td>
</tr>
<tr>
<td>cols</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of columns of the dense matrix</td>
</tr>
<tr>
<td>ld</td>
<td>HOST</td>
<td>OUT</td>
<td>Leading dimension of the dense matrix</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense matrix. Array of size ld * cols</td>
</tr>
<tr>
<td>valueType</td>
<td>HOST</td>
<td>OUT</td>
<td>Enumerator specifying the datatype of values</td>
</tr>
<tr>
<td>order</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()

```
cusparseStatus_t
cusparseDnMatGetValues(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>dnMatDescr</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix descriptor</td>
</tr>
<tr>
<td>values</td>
<td>DEVICE</td>
<td>OUT</td>
<td>Values of the dense matrix. Array of size ld * cols</td>
</tr>
</tbody>
</table>

See `cusparseStatus_t` for the description of the return status

14.5.5. cusparseDnSetValues()

```
cusparseStatus_t
cusparseDnSetValues(cusparseDnMatDescr_t dnMatDescr,
                    void*                values)
```

This function sets the `values` field of the dense matrix descriptor `dnMatDescr`.

See `cusparseStatus_t` for the description of the return status
14.5.6. **cusparseDnMatGetStridedBatch()**

```c
cusparseStatus_t
cusparseDnMatGetStridedBatch(cusparseDnMatDescr_t dnMatDescr,
int* batchCount,
int64_t* batchStride)
```

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

14.5.7. **cusparseDnMatSetStridedBatch()**

```c
cusparseStatus_t
cusparseDnMatSetStridedBatch(cusparseDnMatDescr_t dnMatDescr,
int batchCount,
int64_t 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. batchStride ≥ ld * cols if the matrix uses column-major layout, batchStride ≥ ld * rows otherwise</td>
</tr>
</tbody>
</table>

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

14.6. **Generic API Functions**
14.6.1. cusparseSparseToDense()

The function converts the sparse matrix \( \text{matA} \) in CSR, CSC, or COO format into its dense representation \( \text{matB} \). Blocked-ELL is not currently supported.

The function `cusparseSparseToDense_bufferSize()` returns the size of the workspace needed by `cusparseSparseToDense()`.

<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>matA</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix ( \text{A} )</td>
</tr>
<tr>
<td><code>matB</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Dense matrix ( \text{B} )</td>
</tr>
<tr>
<td><code>alg</code></td>
<td>HOST</td>
<td>IN</td>
<td>Algorithm for the computation</td>
</tr>
<tr>
<td><code>bufferSize</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by <code>cusparseSparseToDense()</code></td>
</tr>
<tr>
<td><code>buffer</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer</td>
</tr>
</tbody>
</table>

`cusparseSparseToDense()` supports the following index type for representing the sparse matrix `matA`:

- 32-bit indices (`CUSPARSE_INDEX_32I`)
- 64-bit indices (`CUSPARSE_INDEX_64I`)

`cusparseSparseToDense()` 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>
cusparseSparse2Dense() supports the following 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>

cusparseSparseToDense() has the following properties:
- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run

cusparseSparseToDense() 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 - cusparseSparseToDense for a code example.

14.6.2. cusparseDenseToSparse()

cusparseStatus_t cusparseDenseToSparse_bufferSize(cusparseHandle_t handle, cusparseDnMatDescr_t matA, cusparseSpMatDescr_t matB, cusparseDenseToSparseAlg_t alg, size_t* bufferSize)

cusparseStatus_t cusparseDenseToSparse_analysis(cusparseHandle_t handle, cusparseDnMatDescr_t matA, cusparseSpMatDescr_t matB, cusparseDenseToSparseAlg_t alg, void* buffer)

cusparseStatus_t cusparseDenseToSparse_convert(cusparseHandle_t handle, cusparseDnMatDescr_t matA, cusparseSpMatDescr_t matB, cusparseDenseToSparseAlg_t alg, void* buffer)

The function converts the dense matrix matA into a sparse matrix matB in CSR, CSC, COO, or Blocked-ELL 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 matB. The user is responsible to allocate the memory required by the sparse matrix:
- Row/Column indices and value arrays for CSC and CSR respectively
- Row, column, value arrays for COO
- Column \((\text{ellColInd})\), value \((\text{ellValue})\) arrays for Blocked-ELL

Finally, we call \texttt{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 (A)</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>OUT</td>
<td>Sparse 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 \texttt{cusparseDenseToSparse_analysis()}</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 \((\text{CUSPARSE\_INDEX\_32I})\)
- 64-bit indices \((\text{CUSPARSE\_INDEX\_64I})\)

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 for CSR, CSC, COO formats.

Hardware Memory Compression

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

Please visit cuSPARSE Library Samples - cusparseDenseToSparse [CSR] and cuSPARSE Library Samples - cusparseDenseToSparse [Blocked-ELL] for code examples.

### 14.6.3. `cusparseAxpby()`

```c

cusparseStatus_t cusparseAxpby(cusparseHandle_t     handle,
                           const void*          alpha,
                           cusparseSpVecDescr_t vecX,
                           const void*          beta,
                           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,

```c
for i=0 to n-1
    Y[i] = beta * Y[i]
for i=0 to nnz-1
    Y[X_indices[i]] += alpha * 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>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\alpha$ scalar used for multiplication</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector $X$</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\beta$ scalar used for multiplication</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense vector $Y$</td>
</tr>
</tbody>
</table>

`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:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>X/Y/compute</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td></td>
</tr>
<tr>
<td>CUDA_C_32F</td>
<td></td>
</tr>
</tbody>
</table>
Mixed-precision computation:

<table>
<thead>
<tr>
<th>x/y</th>
<th>compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td>CUDA_C_32F</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.

### 14.6.4. cusparseGather()

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,

```plaintext```
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x/y</strong></td>
<td></td>
</tr>
<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>

**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.

### 14.6.5. **cusparseScatter()**

**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]
```

### cusparseScatter

<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 `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.

---

### 14.6.6. cusparseRot()

```c
 cusparseStatus_t cusparseRot(cusparseHandle_t handle, const void* c_coeff,
```

---
The function computes the Givens rotation matrix

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

to a sparse vecX and a dense vector vecY

In other words,

for i=0 to nnz-1

\[
\begin{align*}
Y[X_{\text{indices}[i]]} &= c \times Y[X_{\text{indices}[i]}] - s \times X_{\text{values}[i]} \\
X_{\text{values}[i]} &= c \times X_{\text{values}[i]} + s \times Y[X_{\text{indices}[i]}]
\end{align*}
\]

<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 vecX:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)

cusparseRot supports the following datatypes:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>X/Y/compute</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td></td>
</tr>
<tr>
<td>CUDA_R_64F</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>X/Y</th>
<th>compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td>CUDA_C_32F</td>
</tr>
</tbody>
</table>

cusparseRot() has the following constraints:

- The arrays representing the sparse vector vecX must be aligned to 16 bytes
cuSPARSE Generic API Reference

14.6.7. cuSparseSpVV()  

The function computes the inner dot product of a sparse vector vecX and a dense vector vecY:

\[ \text{result} = X' \cdot Y \]

In other words,

\[
\begin{align*}
\text{result} &= 0; \\
\text{for } i = 0 \text{ to } \text{nnz-1} & \\
\text{result} &= \text{result } + X_{\text{values}[i]} \times Y[X_{\text{indices}[i]}]
\end{align*}
\]

\[
op(X) = \begin{cases} X & \text{if op(X) == CUSPARSE_OPERATION__NON_TRANSPOSE} \\ X^* & \text{if op(X) == CUSPARSE_OPERATION__CONJUGATE_TRANSPOSE} \end{cases}
\]

The function cuSparseSpVV_bufferSize() returns the size of the workspace needed by 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>
</tbody>
</table>
### cuSPARSE Generic API Reference

#### cuSPARSE Library

<table>
<thead>
<tr>
<th>Param.</th>
<th>Memory</th>
<th>In/out</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>opX</code></td>
<td>HOST</td>
<td>IN</td>
<td>Operation <code>op(X)</code> that is non-transpose or conjugate transpose</td>
</tr>
<tr>
<td><code>vecX</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse vector <code>X</code></td>
</tr>
<tr>
<td><code>vecY</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector <code>Y</code></td>
</tr>
<tr>
<td><code>result</code></td>
<td>HOST or DEVICE</td>
<td>OUT</td>
<td>The resulting dot product</td>
</tr>
<tr>
<td><code>computeType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Datatype in which the computation is executed</td>
</tr>
<tr>
<td><code>bufferSize</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by <code>cusparseSpVV</code></td>
</tr>
<tr>
<td><code>externalBuffer</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least <code>bufferSize</code> bytes</td>
</tr>
</tbody>
</table>

`cusparseSpVV` supports the following index type for representing the sparse vector `vecX`:

- 32-bit indices (`CUSPARSE_INDEX_32I`)
- 64-bit indices (`CUSPARSE_INDEX_64I`)

The datatypes combinations currently supported for `cusparseSpVV` are listed below:

#### Uniform-precision computation:

<table>
<thead>
<tr>
<th><code>X/Y</code>/<code>computeType</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</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><code>X/Y</code></th>
<th><code>computeType/result</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_32I</td>
</tr>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_C_16F</td>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_16BF</td>
<td>CUDA_C_32F</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.

14.6.8. cusparseSpMV()

```
cusparseStatus_t
cusparseSpMV_bufferSize(cusparseHandle_t handle,
                        cusparseOperation_t opA,
                        const void* alpha,
                        cusparseSpMatDescr_t matA,
                        cusparseDnVecDescr_t vecX,
                        const void* beta,
                        cusparseDnVecDescr_t vecY,
                        cudaDataType computeType,
                        cusparseSpMVAlg_t alg,
                        size_t* bufferSize)
```

```
cusparseStatus_t
cusparseSpMV(cusparseHandle_t handle,
            cusparseOperation_t opA,
            const void* alpha,
            cusparseSpMatDescr_t matA,
            cusparseDnVecDescr_t vecX,
            const void* beta,
            cusparseDnVecDescr_t vecY,
            cudaDataType computeType,
            cusparseSpMVAlg_t alg,
            void* externalBuffer)
```

This function performs the multiplication of a sparse matrix matA and a dense vector vecX

\[ Y = \alpha op(A) \cdot X + \beta Y \]

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
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>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 op(A)</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\alpha$ scalar used for multiplication of type <code>computeType</code></td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix $A$</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector $X$</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>$\beta$ scalar used for multiplication of type <code>computeType</code></td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense vector $Y$</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Datatype in which the computation is executed</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>cusparseSpMV</code></td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least <code>bufferSize</code> bytes</td>
</tr>
</tbody>
</table>

The sparse matrix formats currently supported are listed below:

- `CUSPARSE_FORMAT_COO`
- `CUSPARSE_FORMAT_COO_AOS [deprecated]`
- `CUSPARSE_FORMAT_CSR`

`cusparseSpMV` supports the following index type for representing the sparse matrix `matA`:

- 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/ Y/computeType</th>
<th></th>
</tr>
</thead>
<tbody>
<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>
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_32I</td>
<td>CUDA_R_32I</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_16BF</td>
<td>CUDA_R_16BF</td>
</tr>
</tbody>
</table>

Mixed Regular/Complex computation:

<table>
<thead>
<tr>
<th>(A)</th>
<th>(X/Y)/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseSpMV() supports the following algorithms:

[D]: Deprecated

<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_SPMV_ALG_DEFAULT</td>
<td>Default algorithm for COO sparse matrix format. May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_SPMV_COO_ALG1</td>
<td>Provides deterministic [bit-wise] results for each run. If (\text{opA} \neq \text{CUSPARSE_OPERATION_NON_TRANSPOSE}), it is identical to CUSPARSE_SPMV_COO_ALG1</td>
</tr>
<tr>
<td>CUSPARSE_CSRMV_ALG1</td>
<td>Default algorithm for CSR sparse matrix format. May produce slightly different results during different runs with the same input parameters</td>
</tr>
<tr>
<td>CUSPARSE_CSRMV_ALG2</td>
<td>Provides deterministic [bit-wise] results for each run. If (\text{opA} \neq \text{CUSPARSE_OPERATION_NON_TRANSPOSE}), it is identical to CUSPARSE_SPMV_COO_ALG1</td>
</tr>
</tbody>
</table>

Performance notes:

- CUSPARSE_SPMV_COO_ALG1 and CUSPARSE_SPMV_CSR_ALG1 provide higher performance than CUSPARSE_SPMV_COO_ALG2 and CUSPARSE_SPMV_CSR_ALG2.
- In general, \(\text{opA} == \text{CUSPARSE_OPERATION_NON_TRANSPOSE}\) is 3x faster than \(\text{opA} != \text{CUSPARSE_OPERATION_NON_TRANSPOSE}\).

cusparseSpMV() has the following properties:

- The routine requires extra storage for CSR format (all algorithms) and for COO format with CUSPARSE_SPMV_COO_ALG2 algorithm.
- Provides deterministic (bit-wise) results for each run only for CUSPARSE_SPMM_COO_ALG2 and CUSPARSE_SPMM_CSR_ALG2 algorithms, and \( \text{opA} = \text{CUSPARSE_OPERATION_NON_TRANSPOSE} \).
- The routine supports asynchronous execution.

\texttt{cusparseSpMV()} supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

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

Please visit \texttt{cuSPARSE Library Samples - cusparseSpMV CSR} and \texttt{cusparseSpMV COO} for a code example.

### 14.6.9. \texttt{cusparseSpSV()}

```c
cusparseStatus_t
cusparseSpSV_createDescr(cusparseSpSVDescr_t* spsvDescr);

cusparseStatus_t
cusparseSpSV_destroyDescr(cusparseSpSVDescr_t spsvDescr);

cusparseStatus_t
cusparseSpSV_bufferSize(cusparseHandle_t     handle,
                        cusparseOperation_t  opA,
                        const void*          alpha,
                        cusparseSpMatDescr_t matA,
                        cusparseDnVecDescr_t vecX,
                        cusparseDnVecDescr_t vecY,
                        cudaDataType         computeType,
                        cusparseSpSVAlg_t    alg,
                        cusparseSpSVDescr_t  spsvDescr,
                        size_t*              bufferSize)

cusparseStatus_t
cusparseSpSV_analysis(cusparseHandle_t     handle,
                      cusparseOperation_t  opA,
                      const void*          alpha,
                      cusparseSpMatDescr_t matA,
                      cusparseDnVecDescr_t vecX,
                      cusparseDnVecDescr_t vecY,
                      cudaDataType         computeType,
                      cusparseSpSVAlg_t    alg,
                      cusparseSpSVDescr_t  spsvDescr
                      void*                externalBuffer)

cusparseStatus_t
cusparseSpSV_solve(cusparseHandle_t     handle,
                   cusparseOperation_t  opA,
                   const void*          alpha,
                   cusparseSpMatDescr_t matA,
                   cusparseDnVecDescr_t vecX,
                   cusparseDnVecDescr_t vecY,
                   cudaDataType         computeType,
                   cusparseSpSVAlg_t    alg,
                   void*                externalBuffer)
```
The function solves a system of linear equations whose coefficients are represented in a sparse triangular matrix:

\[ op(A) \cdot Y = \alpha X \]

where

- \( op(A) \) is a sparse square matrix of size \( m \times m \)
- \( X \) is a dense vector of size \( m \)
- \( Y \) is a dense vector of size \( m \)
- \( \alpha \) is a scalar

Also, for matrix \( A \)

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

The function `cusparseSpSV_bufferSize()` returns the size of the workspace needed by `cusparseSpSV_analysis()` and `cusparseSpSV_solve()`. The function `cusparseSpSV_analysis()` performs the analysis phase, while `cusparseSpSV_solve()` executes the solve phase for a sparse triangular linear system. The opaque data structure `spsvDescr` is used to share information among all functions.

The routine supports arbitrary sparsity for the input matrix, but only the upper or lower triangular part is taken into account in the 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 ( op(A) )</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>( \alpha ) scalar used for multiplication of type <code>computeType</code></td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix ( A )</td>
</tr>
<tr>
<td>vecX</td>
<td>HOST</td>
<td>IN</td>
<td>Dense vector ( X )</td>
</tr>
<tr>
<td>vecY</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense vector ( Y )</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Datatype in which the computation is executed</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>cusparseSpSV_analysis()</code> and <code>cusparseSpSV_solve()</code></td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least <code>bufferSize</code> bytes. It is used by <code>cusparseSpSV_analysis</code> and <code>cusparseSpSV_solve()</code></td>
</tr>
<tr>
<td>spsvDescr</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Opaque descriptor for storing internal data used across the three steps</td>
</tr>
</tbody>
</table>
The sparse matrix formats currently supported are listed below:

- CUSPARSE_FORMAT_CSR
- CUSPARSE_FORMAT_COO

The `cusparseSpSV()` supports the following shapes and properties:

- CUSPARSE_FILL_MODE_LOWER and CUSPARSE_FILL_MODE_UPPER fill modes
- CUSPARSE_DIAG_TYPE_NON_UNIT and CUSPARSE_DIAG_TYPE_UNIT diagonal types

The fill mode and diagonal type can be set by `cusparseSpMatSetAttribute()`

`cusparseSpSV()` supports the following index type for representing the sparse matrix `matA`:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)

`cusparseSpSV()` supports the following datatypes:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>A/ X / Y / computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

`cusparseSpSV()` supports the following algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SPSV_ALG_DEFAULT</td>
<td>Default algorithm</td>
</tr>
</tbody>
</table>

`cusparseSpSV()` has the following properties:

- The routine requires extra storage for the analysis phase which is proportional to number of non-zero entries of the sparse matrix
- Provides deterministic (bit-wise) results for each run for the solving phase `cusparseSpSV_solve()`
- The routine supports asynchronous execution

`cusparseSpSV()` 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 - cusparseSpSV CSR and cuSPARSE Library Samples - cusparseSpSV COO for code examples.
14.6.10.  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_preprocess(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)
```

```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`

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

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 `matB` and a sparse matrix `matA` by switching the dense matrices layout:
\[ C_C = B_C \cdot A + \beta C_C \]
\[ C_R = A^T \cdot B_R + \beta C_R \]

where \( B_C, C_C \) indicate column-major layout, while \( B_R, C_R \) refer to row-major layout.

Also, for matrix \( A \) and \( B \)

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

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 \texttt{cusparseSpMM\_bufferSize()} returns the size of the workspace needed by \texttt{cusparseSpMM()}.

The function \texttt{cusparseSpMM\_preprocess()} can be called before \texttt{cusparseSpMM} to speed up the actual computation. It is useful when \texttt{cusparseSpMM} is called multiple times with the same sparsity pattern (\texttt{matA}). The values of the dense matrices (\texttt{matB}, \texttt{matC}) can change arbitrarily. It requires \texttt{CUSPARSE\_SPMM\_CSR\_ALG3}. All other formats and algorithm have no effect.

<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 of type \text{computeType}</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 of type \text{computeType}</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>Datatype in which the computation is executed</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 \texttt{cusparseSpMM}</td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer of at least bufferSize bytes</td>
</tr>
</tbody>
</table>

cusparseSpMM supports the following sparse matrix formats:
cuSPARSE Generic API Reference

- CUSPARSE_FORMAT_COO
- CUSPARSE_FORMAT_CSR
- CUSPARSE_FORMAT_BLOCKED_ELL

(1) COO/CSR FORMATS

cusparseSpMM supports the following index type for representing the sparse matrix matA:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I) only with CUSPARSE_SPMM_COO_ALG4, CUSPARSE_SPMM_CSR_ALG2, and CUSPARSE_SPMM_CSR_ALG3 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_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_16BF</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td>CUDA_R_16BF</td>
<td>CUDA_R_16BF</td>
<td>CUDA_R_16BF</td>
</tr>
</tbody>
</table>

NOTE 1: CUDA_R_16BF data type is supported only with CUSPARSE_SPMM_COO_ALG4, CUSPARSE_SPMM_CSR_ALG2, and CUSPARSE_SPMM_CSR_ALG3 algorithms.

NOTE 2: CUDA_C_16BF data type is supported only with CUSPARSE_SPMM_COO_ALG4 and CUSPARSE_SPMM_CSR_ALG2.

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 [D]</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_ALG2 [D]</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></td>
<td>▶ It provides deterministic result</td>
</tr>
<tr>
<td></td>
<td>▶ It requires additional memory</td>
</tr>
<tr>
<td>CUSPARSE_COOMM_ALG3 [D]</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_CSRMM_ALG1 [D]</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_CSRMM_ALG2 [D]</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>
<tr>
<td>CUSPARSE_CSRMM_ALG3 [D]</td>
<td>Algorithm 3 for CSR sparse matrix format</td>
</tr>
</tbody>
</table>
Algorithm | Notes
--- | ---
| | ‣ It supports only `opA == CUSPARSE_OPERATION_NON_TRANSPOSE` (fallback to `CUSPARSE_SPMM_CSR_ALG2`)  
| | ‣ It does not support CUDA_C_16F and CUDA_C_16BF data types

**Performance notes:**

- 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:

- $C_i = A_i \cdot B_i$
- $C_i = A_i \cdot B$
- $C_i = A_i \cdot B_i$

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`

`cusparseSpMM()` supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

Please visit **cuSPARSE Library Samples - cusparseSpMM CSR** and **cusparseSpMM COO** for a code example.

### BLOCKED-ELLPACK FORMAT

`cusparseSpMM` supports the following datatypes for `CUSPARSE_FORMAT_BLOCKED_ELL` format and the following GPU architectures for exploiting NVIDIA Tensor Cores:

<table>
<thead>
<tr>
<th>A/B</th>
<th>C</th>
<th><code>computeType</code></th>
<th><code>opB</code></th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
<td>N, T</td>
<td>≥ 70</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>N, T</td>
<td>≥ 70</td>
</tr>
<tr>
<td>A/B</td>
<td>C</td>
<td>computeType</td>
<td>opB</td>
<td>Compute Capability</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------------------</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>N,T</td>
<td>≥ 70</td>
</tr>
<tr>
<td>CUDA_R_8I</td>
<td>CUDA_R_8I</td>
<td>CUDA_R_32I</td>
<td>N</td>
<td>≥ 75</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>N,T</td>
<td>≥ 80</td>
</tr>
<tr>
<td>CUDA_R_16F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>N,T</td>
<td>≥ 80</td>
</tr>
<tr>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>CUDA_R_32F</td>
<td>N,T</td>
<td>≥ 80</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>CUDA_R_64F</td>
<td>N,T</td>
<td>≥ 80</td>
</tr>
</tbody>
</table>

cusparseSpMM supports the following algorithms with CUSPARSE_FORMAT_BLOCKED_ELL format:

[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></td>
</tr>
<tr>
<td>CUSPARSE_SPMM_BLOCKED_DEFAULT</td>
<td>Default algorithm for Blocked-ELL format</td>
</tr>
</tbody>
</table>

Performance notes:

- Blocked-ELL SpMM provides the best performance with Power-of-2 Block-Sizes
- Large Block-Sizes (e.g. ≥ 64) provide the best performance

The function has the following limitations:

- The pointer mode must be equal to CUSPARSE_POINTER_MODE_HOST
- Only opA == CUSPARSE_OPERATION_NON_TRANSPOSE is supported
- opB == CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE is not supported

Please visit cuSPARSE Library Samples - cusparseSpMM Blocked-ELL for a code example.

See cusparseStatus_t for the description of the return status

14.6.11. cusparseSpSM()
The function solves a system of linear equations whose coefficients are represented in a sparse triangular matrix:

\[ \text{op}(A) \cdot C = \alpha \text{op}(B) \]

where

- \( \text{op}(A) \) is a sparse square matrix of size \( m \times m \)
- \( \text{op}(B) \) is a dense matrix of size \( m \times n \)
- \( C \) is a dense matrix of size \( m \times n \)
- \( \alpha \) is a scalar

Also, for matrix \( \lambda \)

\[ \text{op}(A) = \begin{cases} A & \text{if op}(A) = \text{CUSPARSE_ OPERATION_NON_TRANSPOSE} \\ A^T & \text{if op}(A) = \text{CUSPARSE_ OPERATION_TRANSPOSE} \\ A^H & \text{if op}(A) = \text{CUSPARSE_ OPERATION_CONJUGATE_TRANSPOSE} \end{cases} \]

\[ \text{op}(B) = \begin{cases} B & \text{if op}(B) = \text{CUSPARSE_ OPERATION_NON_TRANSPOSE} \\ B^T & \text{if op}(B) = \text{CUSPARSE_ OPERATION_TRANSPOSE} \end{cases} \]
The function `cusparseSpSM_bufferSize()` returns the size of the workspace needed by `cusparseSpSM_analysis()` and `cusparseSpSM_solve()`. The function `cusparseSpSM_analysis()` performs the analysis phase, while `cusparseSpSM_solve()` executes the solve phase for a sparse triangular linear system. The opaque data structure `spsmDescr` is used to share information among all functions.

The routine supports arbitrary sparsity for the input matrix, but only the upper or lower triangular part is taken into account in the computation.

`cusparseSpSM_bufferSize()` requires a buffer size for the analysis phase which is proportional to number of non-zero entries of the sparse matrix.

<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 op(A)</td>
</tr>
<tr>
<td><code>opB</code></td>
<td>HOST</td>
<td>IN</td>
<td>Operation op(B)</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>(\alpha) scalar used for multiplication of type <code>computeType</code></td>
</tr>
<tr>
<td><code>matA</code></td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix A</td>
</tr>
<tr>
<td><code>matB</code></td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix B</td>
</tr>
<tr>
<td><code>matC</code></td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Dense matrix C</td>
</tr>
<tr>
<td><code>computeType</code></td>
<td>HOST</td>
<td>IN</td>
<td>Datatype in which the computation is executed</td>
</tr>
<tr>
<td><code>alg</code></td>
<td>HOST</td>
<td>IN</td>
<td>Algorithm for the computation</td>
</tr>
<tr>
<td><code>bufferSize</code></td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by <code>cusparseSpSM_analysis()</code> and <code>cusparseSpSM_solve()</code></td>
</tr>
<tr>
<td><code>externalBuffer</code></td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least <code>bufferSize</code> bytes. It is used by <code>cusparseSpSM_analysis</code> and <code>cusparseSpSM_solve()</code></td>
</tr>
<tr>
<td><code>spsmDescr</code></td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Opaque descriptor for storing internal data used across the three steps</td>
</tr>
</tbody>
</table>

The sparse matrix formats currently supported are listed below:

- CUSPARSE_FORMAT_CSR
- CUSPARSE_FORMAT_COO

The `cusparseSpSM()` supports the following shapes and properties:

- CUSPARSE_FILL_MODE_LOWER and CUSPARSE_FILL_MODE_UPPER fill modes
- CUSPARSE_DIAG_TYPE_NON_UNIT and CUSPARSE_DIAG_TYPE_UNIT diagonal types

The fill mode and diagonal type can be set by `cusparseSpMatSetAttribute()`.

`cusparseSpSM()` supports the following index type for representing the sparse matrix `matA`:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)
cusparseSpSM() supports the following datatypes:

Uniform-precision computation:

<table>
<thead>
<tr>
<th>A/B/C/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseSpSM() supports the following algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SPSM_ALG_DEFAULT</td>
<td>Default algorithm</td>
</tr>
</tbody>
</table>

cusparseSpSM() has the following properties:

- The routine requires no extra storage
- Provides deterministic (bit-wise) results for each run for the solving phase cusparseSpSM_solve()
- The routine supports asynchronous execution

cusparseSpSM() 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 - cusparseSpSM CSR and cuSPARSE Library Samples - cusparseSpSM COO for code examples.

14.6.12. cusparseConstrainedGeMM() [DEPRECATED]

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

cusparseStatus_t

```c
 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)
```

```c
 cusparseStatus_t

 cusparseConstrainedGeMM_bufferSize(cusparseHandle_t handle,
```
The function has the same behavior of `cusparseSDDMM()`. 

14.6.13. `cusparseSDDMM()`

```c
cusparseStatus_t cusparseSDDMM_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,
size_t* bufferSize)
```

```c
cusparseStatus_t cusparseSDDMM_preprocess(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,
cusparseSDDMMAlg_t alg,
void* externalBuffer)
```

```c
cusparseStatus_t cusparseSDDMM(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,
cusparseSDDMMAlg_t alg,
void* externalBuffer)
```

This function performs the multiplication of `matA` and `matB`, followed by an element-wise multiplication with the sparsity pattern of `matC`. Formally, it performs the following operation:

\[ C = \alpha(op(A) \cdot op(B)) \circ \text{spy}(C) + \beta C \]
where

- \( \text{op}(A) \) is a dense matrix of size \( m \times k \)
- \( \text{op}(B) \) 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:

\[
\text{spy}(C)_{ij} = \begin{cases} 
0 & \text{if } C_{ij} = 0 \\
1 & \text{otherwise}
\end{cases}
\]

Also, for matrix \( A \) and \( B \)

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

\text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} is currently not supported.

The function \text{cusparseSDDMM\_bufferSize()} returns the size of the workspace needed by \text{cusparseSDDMM} or \text{cusparseSDDMM\_preprocess}.

The function \text{cusparseSDDMM\_preprocess()} can be called before \text{cusparseSDDMM} to speedup the actual computation. It is useful when \text{cusparseSDDMM} is called multiple times with the same sparsity pattern (\text{matC}). The values of the dense matrices (\text{matA}, \text{matB}) can change arbitrarily.

<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 op(A)</td>
</tr>
<tr>
<td>opB</td>
<td>HOST</td>
<td>IN</td>
<td>Operation op(B)</td>
</tr>
<tr>
<td>alpha</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>( \alpha ) scalar used for multiplication of type computeType</td>
</tr>
<tr>
<td>matA</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix matA</td>
</tr>
<tr>
<td>matB</td>
<td>HOST</td>
<td>IN</td>
<td>Dense matrix matB</td>
</tr>
<tr>
<td>beta</td>
<td>HOST or DEVICE</td>
<td>IN</td>
<td>( \beta ) scalar used for multiplication of type computeType</td>
</tr>
<tr>
<td>matC</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Sparse matrix matC</td>
</tr>
<tr>
<td>computeType</td>
<td>HOST</td>
<td>IN</td>
<td>Datatype in which the computation is executed</td>
</tr>
<tr>
<td>alg</td>
<td>HOST</td>
<td>IN</td>
<td>Algorithm for the computation</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>bufferSize</td>
<td>HOST</td>
<td>OUT</td>
<td>Number of bytes of workspace needed by cusparseSDDMM</td>
</tr>
<tr>
<td>externalBuffer</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to a workspace buffer of at least bufferSize bytes</td>
</tr>
</tbody>
</table>

Currently supported sparse matrix formats:

- CUSPARSE_FORMAT_CSR

cusparseSpSV() supports the following index type for representing the sparse matrix matA:

- 32-bit indices (CUSPARSE_INDEX_32I)
- 64-bit indices (CUSPARSE_INDEX_64I)

The datatypes combinations currently supported for cusparseSDDMM are listed below:

**Uniform-precision computation:**

<table>
<thead>
<tr>
<th>A/X/ Y/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA_R_32F</td>
</tr>
<tr>
<td>CUDA_R_64F</td>
</tr>
<tr>
<td>CUDA_C_32F</td>
</tr>
<tr>
<td>CUDA_C_64F</td>
</tr>
</tbody>
</table>

cusparseSpSV() supports the following algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSPARSE_SDDMM_ALG_DEFAULT</td>
<td>Default algorithm</td>
</tr>
</tbody>
</table>

cusparseSDDMM() has the following properties:

- The routine requires no extra storage
- Provides deterministic (bit-wise) results for each run
- The routine supports asynchronous execution

cusparseSDDMM() 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 - cusparseSDDMM for a code example.

**14.6.14. cusparseSpGEMM()**

```c

cusparseStatus_t
cusparseSpGEMM_createDescr(cusparseSpGEMMDescr_t* descr)
```
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.

The functions `cusparseSpGEMM_workEstimation()` and `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>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>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>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 requested by <code>cusparseSpGEMM_workEstimation</code></td>
</tr>
<tr>
<td>bufferSize2</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace requested by <code>cusparseSpGEMM_compute</code></td>
</tr>
<tr>
<td>externalBuffer1</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by <code>cusparseSpGEMM_workEstimation</code> and <code>cusparseSpGEMM_compute</code></td>
</tr>
<tr>
<td>externalBuffer2</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by <code>cusparseSpGEMM_compute</code> and <code>cusparseSpGEMM_copy</code></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 return `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 $\text{opA}$, $\text{opB}$ 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></td>
<td>CUDA_R_16F</td>
</tr>
<tr>
<td></td>
<td>CUDA_R_16BF</td>
</tr>
<tr>
<td></td>
<td>CUDA_R_32F</td>
</tr>
</tbody>
</table>
### A/B/C/computeType

<table>
<thead>
<tr>
<th>A/B/C/computeType</th>
</tr>
</thead>
<tbody>
<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>

The cuSPARSE SpGEMM 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>

The cuSPARSE SpGEMM() function has the following properties:

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

cuSPARSE SpGEMM() 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.

### 14.6.15. cusparseSpGEMMreuse() 

cusparseStatus_t cusparseSpGEMM_createDescr(cusparseSpGEMMDescr_t* descr)

cusparseStatus_t cusparseSpGEMM_destroyDescr(cusparseSpGEMMDescr_t descr)

cusparseStatus_t cusparseSpGEMMreuse_workEstimation(cusparseHandle_t handle,
                                                  cusparseOperation_t opA,
                                                  cusparseOperation_t opB,
                                                  cusparseSpMatDescr_t matA,
                                                  cusparseSpMatDescr_t matB,
                                                  cusparseSpMatDescr_t matC,
                                                  cusparseSpGEMMAlg_t alg,
                                                  cusparseSpGEMMDescr_t spgemmDescr,
                                                  size_t* bufferSize1,
                                                  void* externalBuffer1)

cusparseStatus_t cusparseSpGEMMreuse_nnz(cusparseHandle_t handle,
                                          cusparseOperation_t opA,
                                          cusparseOperation_t opB,
                                          cusparseSpMatDescr_t matA,
This function performs the multiplication of two sparse matrices `matA` and `matB` where the structure of the output matrix `matC` can be reused for multiple computations with different values.

\[ C = \alpha op(A) \cdot op(B) + \beta C \]

where \( \alpha \) and \( \beta \) are scalars.

The functions `cusparseSpGEMMreuse_workEstimation()`, `cusparseSpGEMMreuse_nnz()`, and `cusparseSpGEMMreuse_copy()` are used for 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 op(A)</td>
</tr>
<tr>
<td>opB</td>
<td>HOST</td>
<td>IN</td>
<td>Operation 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>matB</td>
<td>HOST</td>
<td>IN</td>
<td>Sparse matrix B</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>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 requested by cusparseSpGEMMreuse_workEstimation</td>
</tr>
<tr>
<td>bufferSize2</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace requested by cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>bufferSize3</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace requested by cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>bufferSize4</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace requested by cusparseSpGEMMreuse_copy</td>
</tr>
<tr>
<td>bufferSize5</td>
<td>HOST</td>
<td>IN/OUT</td>
<td>Number of bytes of workspace requested by cusparseSpGEMMreuse_copy</td>
</tr>
<tr>
<td>externalBuffer1</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMMreuse_workEstimation and cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>externalBuffer2</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>externalBuffer3</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>externalBuffer4</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMMreuse_nnz</td>
</tr>
<tr>
<td>externalBuffer5</td>
<td>DEVICE</td>
<td>IN</td>
<td>Pointer to workspace buffer needed by cusparseSpGEMMreuse_nnz</td>
</tr>
</tbody>
</table>

**MEMORY REQUIREMENT:** cusparseSpGEMMreuse requires to keep in memory all intermediate products to reuse the structure of the output matrix. On the other hand, the number of intermediate products is orders of magnitude higher than the number of non-zero entries in general. In order to minimize the memory requirements, the routine uses multiple buffers that can be deallocated after they are no more needed. If the number of intermediate product exceeds $2^{31}-1$, 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 `cusparseSpGEMMreuse` are listed below:

**Uniform-precision computation:**

<table>
<thead>
<tr>
<th>A/B / C/computeType</th>
</tr>
</thead>
<tbody>
<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 / C/computeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA R_16F / CUDA R_32F</td>
</tr>
<tr>
<td>CUDA R_16BF / CUDA R_32F</td>
</tr>
</tbody>
</table>

`cusparseSpGEMMreuse` 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) structure for the output matrix for each run, while value computation is not deterministic</td>
</tr>
<tr>
<td>CUSPARSE_SPGEMM_CSR_ALG_NONDETERMINISTIC</td>
<td>Provides deterministic (bit-wise) structure for the output matrix and value computation for each run</td>
</tr>
</tbody>
</table>

`cusparseSpGEMMreuse()` has the following properties:

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

`cusparseSpGEMMreuse()` supports the following **optimizations**:

- Hardware Memory Compression

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

Please visit cuSPARSE Library Samples - cusparseSpGEMMreuse 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()`, `cudaMemset`, and so on) that can be used to allocate memory on the GPU.

The cuSPARSE Fortran wrapper code is provided as an example only and needs to be compiled into an application for it to call the cuSPARSE API functions. Providing this source code allows users to make any changes necessary for a particular platform and toolchain.

The cuSPARSE Fortran wrapper code has been used to demonstrate interoperability with the compilers g95 0.91 (on 32-bit and 64-bit Linux) and g95 0.92 (on 32-bit and 64-bit Mac OS X). In order to use other compilers, users have to make any changes to the wrapper code that may be required.

The direct wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all cuSPARSE functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPU memory space (using `CUDA_MALLOC()` and `CUDA_FREE()` ) and to copy data between GPU and CPU memory spaces (using the `CUDA_MEMCPY()` routines). The sample wrappers provided in `cusparse_fortran.c` map device pointers to the OS-dependent type `size_t`, which is 32 bits wide on 32-bit platforms and 64 bits wide on a 64-bit platforms.

One approach to dealing with index arithmetic on device pointers in Fortran code is to use C-style macros and to use the C preprocessor to expand them. On Linux and Mac OS X, preprocessing can be done by using the option `-cpp` with g95 or gfortran. The function
GET_SHIFTED_ADDRESS(), provided with the cuSPARSE Fortran wrappers, can also be used, as shown in example B.

Example B shows the the C++ of example A implemented in Fortran 77 on the host. This example should be compiled with ARCH_64 defined as 1 on a 64-bit OS system and as undefined on a 32-bit OS system. For example, on g95 or gfortran, it can be done directly on the command line using the option -cpp -DARCH_64=1.

15.1. 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_dscsr
integer cusparse_dcsrmv
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"

! predefined constants (need to be careful with them)
dzero = 0.0
dtwo = 2.0
dthree = 3.0
dfive = 5.0

! create the following sparse test matrix in COO format
! (notice one-based indexing)
! | 1.0 | 2.0 | 3.0 |
! | 4.0 |     |     |
! | 5.0 | 6.0 | 7.0 |
! | 8.0 | 9.0 |     |
cooRowIndexHostPtr(1)=1
c ooColIndexHostPtr(1)=1
c ooValHostPtr(1) =1.0
c ooRowIndexHostPtr(2)=1
c ooColIndexHostPtr(2)=3
c ooValHostPtr(2) =2.0
c ooRowIndexHostPtr(3)=1
c ooColIndexHostPtr(3)=4
c ooValHostPtr(3) =3.0
c ooRowIndexHostPtr(4)=2
c ooColIndexHostPtr(4)=2
c ooValHostPtr(4) =4.0
c ooRowIndexHostPtr(5)=3
c ooColIndexHostPtr(5)=1
c ooValHostPtr(5) =5.0
c ooRowIndexHostPtr(6)=3
c ooColIndexHostPtr(6)=3
c ooValHostPtr(6) =6.0
c ooRowIndexHostPtr(7)=3
c ooColIndexHostPtr(7)=4
c ooValHostPtr(7) =7.0
c ooRowIndexHostPtr(8)=4
c ooColIndexHostPtr(8)=2
c ooValHostPtr(8) =8.0
c ooRowIndexHostPtr(9)=4
c ooColIndexHostPtr(9)=4
c ooValHostPtr(9) =9.0

! print the matrix
write(*,*) "Input data:"
do i=1,nnz
  write(*,*) "cooRowIndexHostPtr[",i,"]=",cooRowIndexHostPtr(i)
  write(*,*) "cooColIndexHostPtr[",i,"]=",cooColIndexHostPtr(i)
  write(*,*) "cooValHostPtr[",i,"]=",cooValHostPtr(i)
enddo

! create a sparse and dense vector
xVal = [100.0, 200.0, 400.0] (sparse)
xInd = [0, 1, 3] 
y = [10.0, 20.0, 40.0, 50.0, 60.0, 70.0, 80.0] (dense)
(Notice one-based indexing)
yHostPtr(1) = 10.0
yHostPtr(2) = 20.0
yHostPtr(3) = 30.0
yHostPtr(4) = 40.0
yHostPtr(5) = 50.0
yHostPtr(6) = 60.0
yHostPtr(7) = 70.0
yHostPtr(8) = 80.0
xIndHostPtr(1) = 1
xValHostPtr(1) = 100.0
xIndHostPtr(2) = 2
xValHostPtr(2) = 200.0
xIndHostPtr(3) = 4
xValHostPtr(3) = 400.0

print the vectors

allocate GPU memory and copy the matrix and vectors into it

cudaSuccess = 0
cudaMemcpyHostToDevice = 1
cudaStat1 = cuda_malloc(cooRowIndex, nnz*4)
cudaStat2 = cuda_malloc(cooColIndex, nnz*4)
cudaStat3 = cuda_malloc(cooVal, nnz*8)
cudaStat4 = cuda_malloc(y, 2*n*8)
cudaStat5 = cuda_malloc(xInd, nnz_vector*4)
cudaStat6 = cuda_malloc(xVal, nnz_vector*8)

if ((cudaStat1 /= 0) .OR. (cudaStat2 /= 0) .OR. (cudaStat3 /= 0) .OR. (cudaStat4 /= 0) .OR. (cudaStat5 /= 0) .OR. (cudaStat6 /= 0)) then
  write(*,*) "Device malloc failed"
  write(*,*) "cudaStat1=", cudaStat1
  write(*,*) "cudaStat2=", cudaStat2
  write(*,*) "cudaStat3=", cudaStat3
  write(*,*) "cudaStat4=", cudaStat4
  write(*,*) "cudaStat5=", cudaStat5
  write(*,*) "cudaStat6=", cudaStat6
  stop 2
endif

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


```fortran
$  (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)
```

print the matrix descriptor
mtype = cusparse_get_mat_type(descrA)
fn = 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=",fn,"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

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 cuda_free(csrRowPtr)
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 cuda_free(csrRowPtr)
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,
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

print intermediate results (y)
y = [10 20 30 40 | 680 760 1230 2240]
cudaSuccess=0
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
endo
c
exercise Level 3 routines (csrmm)
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_desc(csrA)
call cusparse_destroy(handle)
write(*,*) "Memset on Device failed"
stop 1
endif
status= cusparse_dcsrmm(handle, 0, n, 2, n, nnz, dfive,$                        descrA, cooVal, csrRowPtr, cooColIndex,$                        y, n, dzero, z, n+1)
if (status /= 0) then
  call cuda_free(cooRowIndex)
call cuda_free(cooColIndex)
call cuda_free(cooVal)
call cuda_free(xInd)
call cuda_free(xVal)
call cuda_free(y)
call cuda_free(z)
call cuda_free(csrRowPtr)
call cusparse_destroy_mat_desc(descrA)
call cusparse_destroy(handle)
write(*,*) "Matrix-matrix multiplication failed"
stop 1
endif
c print final results (z)
cudaSuccess=0
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_desc(descrA)
call cusparse_destroy(handle)
write(*,*) "Memcpy from Device to Host failed"
stop 1
endif
c z = [950 400 2550 2600 0 | 49300 15200 132300 131200 0]
write(*,*) "Final results:"
do j=1,2
do i=1,n+1
  write(*,*) "z[",i",",",j",]=",zHostPtr(i+(n+1)*(j-1))
enddo
enddo
c check the results
epsilon = 0.00000000000001
if ((fabs(zHostPtr(1) - 950.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(2) - 400.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(3) - 2550.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(4) - 2600.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(5) - 0.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(6) - 49300.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(7) - 15200.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(8) - 132300.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(9) - 131200.0) .GT. epsilon) .OR.
  (fabs(zHostPtr(10) - 0.0) .GT. epsilon) .OR.
  (fabs(yHostPtr(1) - 10.0) .GT. epsilon) .OR.
  (fabs(yHostPtr(2) - 20.0) .GT. epsilon) .OR.
  (fabs(yHostPtr(3) - 30.0) .GT. epsilon) .OR.
  (fabs(yHostPtr(4) - 40.0) .GT. epsilon) .OR.

$ \text{(DABS(yHostPtr(5) - 680.0) .GT. epsilon) .OR.}$
$\text{(DABS(yHostPtr(6) - 760.0) .GT. epsilon) .OR.}$
$\text{(DABS(yHostPtr(7) - 1230.0) .GT. epsilon) .OR.}$
$\text{(DABS(yHostPtr(8) - 2240.0) .GT. epsilon)) then}$
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,

\[
A = \begin{bmatrix}
1.0 & 2.0 & 0.0 \\
0.0 & 5.0 & 0.0 \\
0.0 & 8.0 & 0.0 \\
\end{bmatrix}
\]

```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;

    /*
    * 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

```c
cudaStat2 = cudaMemcpy(d_cooCols, h_cooCols, sizeof(int)*nnz , cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_cooVals, h_cooVals, sizeof(double)*nnz, cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(d_cooRows, h_cooRows, sizeof(int)*nnz, cudaMemcpyHostToDevice);
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 = cudaMemcpy(d_cooRows, h_cooRows, sizeof(int)*nnz , cudaMemcpyHostToDevice);
cudaStat3 = cudaMemcpy(d_cooCols, h_cooCols, sizeof(int)*nnz , cudaMemcpyHostToDevice);
cudaStat4 = cudaMemcpy(d_P, h_P      , sizeof(int)*nnz , cudaMemcpyHostToDevice);
cudaStat5 = cudaMemcpy(d_cooVals, h_cooVals_sorted, sizeof(double)*nnz, cudaMemcpyHostToDevice);
cudaStat6 = cudaMemcpy(h_cooRows, d_cooRows, sizeof(int)*nnz , cudaMemcpyToDevice);
cudaStat7 = cudaMemcpy(h_cooCols, d_cooCols, sizeof(int)*nnz , cudaMemcpyToDevice);
cudaStat8 = cudaMemcpy(h_P,       d_P      , sizeof(int)*nnz , cudaMemcpyToDevice);
cudaStat9 = cudaMemcpy(h_cooVals, d_cooVals, sizeof(double)*nnz, cudaMemcpyToDevice);

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] );
}
```

cuSPARSE Library
Appendix B: Examples of sorting

/* 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,

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void 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 = %E \n", threshold);
    printMatrix(m, n, A, lda, "A");
    /* step 1: create cusparse handle, bind a stream */
Appendix C: Examples of prune
cuSPARSE Library

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 */

cudaStat1 = cudaMalloc ((void**)&d_A, sizeof(float)*lda*n);
assert(cudaSuccess == cudaStat1);

cudaStat2 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat2);

/* step 3: query workspace */

status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);

cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);

/* step 4: compute csrRowPtrC and nnzC */

status = cusparseSpruneDense2csr_bufferSizeExt(
   handle,
   m,
   n,
   d_A,
   lda,
   &threshold,
   descrC,
   d_csrValC,
   d_csrRowPtrC,
   d_csrColIndC,
   &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);

printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);

if (NULL != d_work) { cudaFree(d_work); }

cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

status = cusparseSpruneDense2csrNnz(
   handle,
   m,
   n,
   d_A,
   lda,
   &threshold,
   descrC,
   d_csrRowPtrC,
   &nnzC, /* host */
   d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);

cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC) {
printf("C is empty \n");
return 0;
}

/* step 5: compute csrColIndC and csrValC */
cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int ) * nnzC );
cudaStat2 = cudaMalloc ((void**)&d_csrValC , sizeof(float) * nnzC );
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);

status = cusparseSpruneDense2csr(
    handle,
    m,
    n,
    d_A,
    lda,
    &threshold,
    descrC,
    d_csrValC,
    d_csrRowPtrC,
    d_csrColIndC,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 6: output C */
csrRowPtrC = (int*) malloc(sizeof(int )*(m+1));
csrColIndC = (int*) malloc(sizeof(int )*nnzC);
csrValC = (float*) malloc(sizeof(float)*nnzC);
assert( NULL != csrRowPtrC);
assert( NULL != csrColIndC);
assert( NULL != csrValC);
cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int )*(m+1),
cudaMemcpyDeviceToHost);
cudaStat2 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int )*nnzC,
cudaMemcpyDeviceToHost);
cudaStat3 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
assert(cudaSuccess == cudaStat2);
assert(cudaSuccess == cudaStat3);

printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
if (d_A) cudaFree(d_A);
if (d_csrRowPtrC) cudaFree(d_csrRowPtrC);
if (d_csrColIndC) cudaFree(d_csrColIndC);
if (d_csrValC) cudaFree(d_csrValC);
if (csrRowPtrC ) free(csrRowPtrC);
if (csrColIndC  ) free(csrColIndC);
if (csrValC     ) free(csrValC);
if (handle      ) cusparseDestroy(handle);
if (stream      ) cudaStreamDestroy(stream);
if (descrC      ) cusparseDestroyMatDescr(descrC);

cudaDeviceReset();
return 0;
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,

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

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

```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;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
```
const int nnzA = 9;
/*
| 1 0 2 -3 |
| 0 4 0 0 |
| 5 0 6 7 |
| 0 8 0 9 |
*/

const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

int* csrRowPtrC = NULL;
int* csrColIndC = NULL;
float* csrValC = NULL;
int *d_csrRowPtrA = NULL;
int *d_csrColIndA = NULL;
float *d_csrValA = NULL;
int *d_csrRowPtrC = NULL;
int *d_csrColIndC = NULL;
float *d_csrValC = NULL;
size_t lworkInBytes = 0;
char *d_work = NULL;

int nnzC = 0;
float threshold = 4.1; /* remove Aij <= 4.1 */
// float threshold = 0; /* remove zeros */
printf("example of pruneCsr2csr \n");
printf("prune |A(i,j)| <= threshold \n");
printf("threshold = %E \n", threshold);

/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A and C */
status = cusparseCreateMatDescr(&descrA);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);
printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");
cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(m+1) );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(float)*nnzA );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1) );
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(m+1),
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(float)*nnzA,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
status = cusparseSpruneCsr2csr_bufferSizeExt(
    handle,
    m, n, nnzA,
    descrA, d_csrValA,
    d_csrRowPtrA, d_csrColIndA,
    &threshold,
    descrC, d_csrValC,
    d_csrRowPtrC, d_csrColIndC,
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes (prune) = %lld \n", (long long)lworkInBytes);
if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
status = cusparseSpruneCsr2csrNnz(
    handle,
    m, n, nnzA,
    descrA, d_csrValA,
    d_csrRowPtrA, d_csrColIndA,
    &threshold,
    descrC, d_csrRowPtrC,
    &nnzC, /* host */
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

printf("nnzC = %d\n", nnzC);
if (0 == nnzC)
{
    printf("C is empty \n");
    return 0;
}

/* step 5: compute csrColIndC and csrValC */
cudaStat1 = cudaMalloc ((void**)&d_csrColIndC, sizeof(int ) * nnzC);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValC , sizeof(float)* nnzC);
assert(cudaSuccess == cudaStat1);

status = cusparseSpruneCsr2csr(
    handle,
    m,
    n,
    nnzA,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    &threshold,
    descrC,
    d_csrValC,
    d_csrRowPtrC,
    d_csrColIndC,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 6: output C */
csrRowPtrC = (int*)malloc(sizeof(int )*(m+1));
csrColIndC = (int*)malloc(sizeof(int )*nnzC);
csrValC = (float*)malloc(sizeof(float)*nnzC);
assert( NULL != csrRowPtrC);
assert( NULL != csrColIndC);
assert( NULL != csrValC);
cudaStat1 = cudaMemcpy(csrRowPtrC, d_csrRowPtrC, sizeof(int  )*(m+1),
    cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(csrColIndC, d_csrColIndC, sizeof(int  )*nnzC ,
    cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(csrValC   , d_csrValC   , sizeof(float)*nnzC ,
    cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
printCsr(m, n, nnzC, descrC, csrValC, csrRowPtrC, csrColIndC, "C");

/* free resources */
if (d_csrRowPtrA ) cudaFree(d_csrRowPtrA);
if (d_csrColIndA ) cudaFree(d_csrColIndA);
if (d_csrValA    ) cudaFree(d_csrValA);
if (d_csrRowPtrC ) cudaFree(d_csrRowPtrC);
if (d_csrColIndC ) cudaFree(d_csrColIndC);
if (d_csrValC    ) cudaFree(d_csrValC);
if (csrRowPtrC   ) free(csrRowPtrC);
if (csrColIndC   ) free(csrColIndC);
if (csrValC      ) free(csrValC);
if (handle       ) cusparseDestroy(handle);
if (stream      ) cudaStreamDestroy(stream);
if (descrA      ) cusparseDestroyMatDescr(descrA);
if (descrC      ) cusparseDestroyMatDescr(descrC);
cudaDeviceReset();
return 0;
Appendix C: Examples of prune

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,

The percentage is 50, which means to prune 50 percent of the dense matrix. The matrix has 16
elements, so 8 out of 16 must be pruned out. Therefore 7 zeros are pruned out, and value 1.0
is also out because it is the smallest among 9 nonzero elements.
/*
* How to compile (assume cuda is installed at /usr/local/cuda/)
*
nvcc -c -I/usr/local/cuda/include prunedense2csrbyP.cpp
*
g++ -o prunedense2csrbyP.cpp prunedense2csrbyP.o -L/usr/local/cuda/lib64 lcusparse -lcudart
*/
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>
void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
for(int row = 0 ; row < m ; row++){
for(int col = 0 ; col < n ; col++){
float Areg = A[row + col*lda];
printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
}
}
}
void printCsr(
int m,
int n,
int nnz,
const cusparseMatDescr_t descrA,
const float *csrValA,
const int *csrRowPtrA,
const int *csrColIndA,
const char* name)
{
const int base = (cusparseGetMatIndexBase(descrA) != CUSPARSE_INDEX_BASE_ONE)?
0:1 ;
printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m, n,
nnz, base);
for(int row = 0 ; row < m ; row++){
const int start = csrRowPtrA[row ] - base;
const int end
= csrRowPtrA[row+1] - base;
for(int colidx = start ; colidx < end ; colidx++){

cuSPARSE Library

DU-06709-001_v11.4 | 314


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;
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);
        }
    }
}
```

```c
int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    ```
Appendix C: Examples of prune
cuSPARSE Library

```
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 |
 |  5  0  6  7 |
 |  0  8  0  9 |
*/

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;
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 *d_csrRowPtrA = NULL;
int *d_csrColIndA = NULL;
float *d_csrValA = NULL;

int *d_csrRowPtrC = NULL;
int *d_csrColIndC = NULL;
float *d_csrValC = NULL;

size_t lworkInBytes = 0;
char *d_work = NULL;

int nnzC = 0;
float percentage = 20; /* remove 20% of nonzeros */

// example of pruneCsr2csrByPercentage

// 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

```c
status = cusparseCreateMatDescr(&descrC);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* C is base-0 */
cusparseSetMatIndexBase(descrC, CUSPARSE_INDEX_BASE_ZERO);
cusparseSetMatType(descrC, CUSPARSE_MATRIX_TYPE_GENERAL);
printCsr(m, n, nnzA, descrA, csrValA, csrRowPtrA, csrColIndA, "A");

cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValA, sizeof(float)*nnzA);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrC, sizeof(int)*(m+1));
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(m+1), cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrValA, csrValA, sizeof(float)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
status = cusparseSpruneCsr2csrByPercentage_bufferSizeExt(
    handle,
    m,
    n,
    nnzA,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    percentage,
    descrC,
    d_csrValC,
    d_csrRowPtrC,
    d_csrColIndC,
    info,
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes = %lld \n", (long long)lworkInBytes);
if (NULL != d_work) { cudaFree(d_work); }
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: compute csrRowPtrC and nnzC */
status = cusparseSpruneCsr2csrNnzByPercentage(
    handle,
    m,
    n,
    nnzA,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    percentage,
    descrC,
    d_csrRowPtrC,
    d_csrColIndC,
    info,
    &nnzC);
assert(CUSPARSE_STATUS_SUCCESS == status);
```

void prune
  (CuMatrix *A, /* 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 gpsi

18.1. Batched Penta-diagonal Solver

This section provides a simple example in the C programming language of gpsiInterleavedBatch.

The example solves two penta-diagonal systems and assumes data layout is NOT interleaved format. Before calling gpsiInterleavedBatch, 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 residual_eval(
    int n,
    const float *ds,
    const float *dl,
    const float *d,
    const float *du,
    const float *dw,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{  
    float r_nrminf = 0;
    for(int i = 0 ; i < n ; i++)
    {
        float dot = 0;
        if (i > 1 )
        {
            dot += ds[i]*x[i-2];
        }
```
if (i > 0 ){
    dot += dl[i]*x[i-1];
}

dot += d[i]*x[i];
if (i < (n-1 )){
    dot += du[i]*x[i+1];
}
if (i < (n-2 ) ){
    dot += dw[i]*x[i+2];
}

float ri = b[i] - dot;
r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);

*r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t cusparseH = NULL;
cublasHandle_t cublasH = NULL;
cudaStream_t stream = NULL;
cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
cudaError_t cudaStat1 = cudaSuccess;

const int n = 4;
const int batchSize = 2;

/*
 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 = cusparseCreate (&cusparseH);
assert(CUSPARSE_STATUS_SUCCESS == status);

status = cusparseSetStream(cusparseH, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: allocate device memory */
cudaStat1 = cudaMalloc ((void**)&d_ds0 , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dl0 , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_d0  , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_du0 , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dw0 , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_ds  , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dl  , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_d   , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_du  , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_dw  , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_B   , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_X   , sizeof(float)*n*batchSize );
assert(cudaSuccess == cudaStat1);

/* step 3: prepare data in device, interleaved format */
cudaStat1 = cudaMemcpy(d_ds0, ds, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_dl0, dl, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_d0 , d , sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_du0, du, sizeof(float)*n*batchSize,
cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_dw0, dw, sizeof(float)*n*batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_B  ,  B, sizeof(float)*n*batchSize, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize();
/* convert ds to interleaved format */
    ds = transpose(ds0) */
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of ds */
    n,  /* number of columns of ds */
    &h_one,
    d_ds0,  /* ds0 is n-by-batchSize */
    n,  /* leading dimension of ds0 */
    &h_zero,
    NULL,
    n,  /* don't care */
    d_ds,  /* ds is batchSize-by-n */
    batchSize);  /* leading dimension of ds */
assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* convert dl to interleaved format */
    dl = transpose(dl0) */
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of dl */
    n,  /* number of columns of dl */
    &h_one,
    d_dl0,  /* dl0 is n-by-batchSize */
    n,  /* leading dimension of dl0 */
    &h_zero,
    NULL,
    n,  /* don't care */
    d_dl,  /* dl is batchSize-by-n */
    batchSize);  /* leading dimension of dl */
assert(CUBLAS_STATUS_SUCCESS == cublasStat);
/* convert d to interleaved format */
    d = transpose(d0) */
cublasStat = cublasSgeam(
cublasH,
    CUBLAS_OP_T, /* transa */
    CUBLAS_OP_T, /* transb, don't care */
    batchSize, /* number of rows of d */
    n,  /* number of columns of d */
    &h_one,
    d_d0,  /* d0 is n-by-batchSize */
    n,  /* leading dimension of d0 */
    &h_zero,
    NULL,
    n,  /* don't care */
    d_d,  /* d is batchSize-by-n */
    batchSize);  /* leading dimension of d */
assert(CUBLAS_STATUS_SUCCESS == cublasStat);
Appendix D: Examples of gpusv

/* 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 = cusparseSgpsvInterleavedBatch_bufferSizeExt(
cusparseH, 
alg0, 
n,
Appendix D: Examples of gsv

```
d_ds,
d_dl,
d_d,
d_du,
d_dw,
d_X,
batchSize,
lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);
printf("lworkInBytes = %lld \n", (long long)lworkInBytes);
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);
/* step 5: solve Aj*xj = bj */
status = cusparseSgpsvInterleavedBatch(
cusparseH,
algo,
n,
d_ds,
d_dl,
d_d,
d_du,
d_dw,
d_X,
batchSize,
d_work);
cudaStat1 = cudaDeviceSynchronize();
assert(CUSPARSE_STATUS_SUCCESS == status);
assert(cudaSuccess == cudaStat1);
/* step 6: convert X back to aggregate format */
/* B = transpose(X) */
cublasStat = cublasSgeam(
cublasH,
CUBLAS_OP_T, /* transa */
CUBLAS_OP_T, /* transb, don't care */
n, /* number of rows of B */
batchSize, /* number of columns of B */
&h_one,
d_X, /* X is batchSize-by-n */
batchSize, /* leading dimension of X */
&h_zero,
NULL, /* 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(
```
n,
ds,
dl,
d,
du,
dw,
B,
X,
&r1_nrminf
);
printf("|b1 - A1*x1| = %E\n", r1_nrminf);

printf("\n==== x2 = inv(A2)*b2 \n");
for(int j = 0 ; j < n; j++){
   printf("x2[%d] = %f\n", j, X[n+j]);
}
float r2_nrminf;
residual_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>

/* compute | b - A*x|_inf */
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;
            const int col = csrColInd[colidx] - base;
            float Aij = csrVal[colidx];
            float xj  = x[col];
```

The code snippet is provided as a simple example to illustrate how to use the `csrsm2` function in cuSparse library. It computes the residual of a triangular system with two right-hand side vectors.

The example demonstrates the use of `cusparseMatDescr_t` for describing the matrix, and it includes checks for the triangular fill mode and the diagonal type. The residual calculation iterates through the matrix rows and computes the dot product of the matrix column with the vector, updating the residual accordingly.

The code also includes comments for compiling and linking the code, which are not shown in the snippet.
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   |
*      |    0     4     5     6   |
*
*  Regard A as a lower triangle matrix L with non-unit diagonal.
*      | 1  5 |
*  Given B = | 2  6 |
*
*      | 1  5 |
*      | 0.5  1.5 |
*      | -0.3333 -3 |
*      | 0 -0.4444 |
*/
const int csrRowPtrA[n+1] = { 1, 4, 5, 8, 10};
const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};
const float B[n*nrhs] = {1,2,3,4,5,6,7,8};
float X[n*nrhs];
int *d_csrRowPtrA = NULL;
int *d_csrColIndA = NULL;
float *d_csrValA = NULL;
float *d_B = NULL;
size_t lworkInBytes = 0;
char *d_work = NULL;

const int algo = 0; /* non-block version */
printf("example of csrsm2 \n");
/* step 1: create cusparse handle, bind a stream */
cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
assert(cudaSuccess == cudaStat1);
status = cusparseCreate(&handle);
assert(CUSPARSE_STATUS_SUCCESS == status);
status = cusparseSetStream(handle, stream);
assert(CUSPARSE_STATUS_SUCCESS == status);

status = cusparseCreateCsrsm2Info(&info);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: configuration of matrix A */
status = cusparseCreateMatDescr(&descrA);
assert(CUSPARSE_STATUS_SUCCESS == status);
/* A is base-1*/
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
/* A is lower triangle */
cusparseSetMatFillMode(descrA, CUSPARSE_FILL_MODE_LOWER);
/* A has non unit diagonal */
cusparseSetMatDiagType(descrA, CUSPARSE_DIAG_TYPE_NON_UNIT);

cudaStat1 = cudaMalloc ((void**)&d_csrRowPtrA, sizeof(int)*(n+1) );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrColIndA, sizeof(int)*nnzA );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_csrValA   , sizeof(float)*nnzA );
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMalloc ((void**)&d_B         , sizeof(float)*n*nrhs );
assert(cudaSuccess == cudaStat1);

cudaStat1 = cudaMemcpy(d_csrRowPtrA, csrRowPtrA, sizeof(int)*(n+1), cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrColIndA, csrColIndA, sizeof(int)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_csrValA   , csrValA   , sizeof(float)*nnzA, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);
cudaStat1 = cudaMemcpy(d_B         , B         , sizeof(float)*n*nrhs, cudaMemcpyHostToDevice);
assert(cudaSuccess == cudaStat1);

/* step 3: query workspace */
status = cusparseScsrsm2_bufferSizeExt( handle,
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n,
    nrhs,
    nnzA,
    &h_one,
    descrA,
    d_csrValA,
    d_csrRowPtrA,
    d_csrColIndA,
    d_B,
    n, /* ldb */
    info,
    policy,
    &lworkInBytes);
assert(CUSPARSE_STATUS_SUCCESS == status);

printf("lworkInBytes = %lld \n", (long long)lworkInBytes);
if (NULL != d_work) { cudaFree(d_work); } 
cudaStat1 = cudaMalloc((void**)&d_work, lworkInBytes);
assert(cudaSuccess == cudaStat1);

/* step 4: analysis */
status = cusparseScsrsm2_analysis(
    handle, 
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n, 
    nrhs, 
    nnzA, 
    &h_one, 
    descrA, 
    d_csrValA, 
    d_csrRowPtrA, 
    d_csrColIndA, 
    d_B, 
    n,   /* ldb */
    info,
    policy,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 5: solve L * X = B */
status = cusparseScsrsm2_solve(
    handle, 
    algo,
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transA */
    CUSPARSE_OPERATION_NON_TRANSPOSE, /* transB */
    n, 
    nrhs, 
    nnzA, 
    &h_one, 
    descrA, 
    d_csrValA, 
    d_csrRowPtrA, 
    d_csrColIndA, 
    d_B, 
    n,   /* ldb */
    info,
    policy,
    d_work);
assert(CUSPARSE_STATUS_SUCCESS == status);
cudaStat1 = cudaDeviceSynchronize();
assert(cudaSuccess == cudaStat1);

/* step 6: measure residual B - A*X */
cudaStat1 = cudaMemcpy(X, d_B, sizeof(float)*n*nrhs, cudaMemcpyDeviceToHost);
assert(cudaSuccess == cudaStat1);
cudaDeviceSynchronize();

printf("==== x1 = inv(A)*b1 \n");
for(int j = 0 ; j < n; j++)
{
    printf("x1[%d] = %f\n", j, X[j]);
}

float r1_nrminf;
residual_eval(
    n, 
    descrA, 
    csrValA, 
    csrRowPtrA, 
    csrColIndA, 
    B, 
    X, 
    &r1_nrminf 
);
printf("|b1 - A*x1| = %E\n", r1_nrminf);
printf("==== x2 = inv(A)*b2 \n");
for(int j = 0 ; j < n; j++){
    printf("x2[%d] = %f\n", j, X[n+j]);
}
float r2_nrminf;
residual_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<t>gtsv implementation is derived from a version developed by Li-Wen Chang from the University of Illinois.
- The cusparse<t>gtsvInterleavedBatch adopts cuThomasBatch developed by Pedro Valero-Lara and Ivan Martínez-Pérez from Barcelona Supercomputing Center and BSC/UPC NVIDIA GPU Center of Excellence.
Chapter 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.

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

© 2007-2021 NVIDIA Corporation & affiliates. All rights reserved.