CUDA

CUSPARSE Library

PG-05329-040_V01
January, 2011

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

Notice

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

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

Trademarks

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

Copyright

Table of Contents

CHAPTER

1

CUSPARSE Library

The NVIDIA® CUDA™ CUSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices and is designed to be called from C or C++. These subroutines can be classified in four categories:

- Level 1 routines include operations between a vector in sparse format and a vector in dense format.
- Level 2 routines include operations between a matrix in sparse format and a vector in dense format.
- Level 3 routines include operations between a matrix in sparse format and a set of vectors (tall matrix) in dense format.
- Conversion routines that allow conversion between different matrix formats.

The library is written using the CUDA parallel programming model and takes advantage of the computational resources of the NVIDIA graphics processor (GPU). The CUSPARSE API assumes that the input and output data reside in GPU (device) memory, not in CPU (host) memory, unless CPU memory is specifically indicated by the string `HostPtr` being part of the parameter name of a function (for example, `*resultHostPtr` in cusparse{S,D,C,Z}doti on page 28).

It is the responsibility of the user to allocate memory and to copy data between GPU memory and CPU memory using standard CUDA runtime API routines, such as, **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemcpy()`**, and **`cudaMemcpyAsync()`**. (The CUDA runtime API is part of the CUDA Toolkit from NVIDIA.) The library is currently designed to run only on single-GPU systems; it does not auto-parallelize across multiple GPUs.

Note: The CUSPARSE library requires hardware with at least 1.1 compute capability. Please see *NVIDIA CUDA C Programming Guide*, Appendix A for the list of all compute capabilities.

CUSPARSE Formats

CUSPARSE supports a Sparse Vector Format and Matrix Formats.

Index Base Format

The library supports zero- and one-based indexing.

Sparse Vector Format

Sparse vectors are represented with two arrays.

- One data array contains all the non-zero values from the equivalent array in dense format.
- One integer index array contains the position of the corresponding non-zero value in the equivalent array in dense format.

For example, this 7×1 dense vector can be stored as a one-based or a zero-based sparse vector.

$$7 \times 1 \text{ vector: } \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.0 & 3.0 & 0.0 & 4.0 \end{bmatrix}$$

$$\text{One-based: } \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1 & 4 & 5 & 7 \end{bmatrix}$$

Zero-based: $\begin{bmatrix} \texttt{1.0} & \texttt{2.0} & \texttt{3.0} & \texttt{4.0} \\ \texttt{0} & \texttt{3} & \texttt{4} & \texttt{6} \end{bmatrix}$

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

Matrix Formats

The matrix formats are the following:

- Dense Format on page 7
- Coordinate Format (COO) on page 8
- Compressed Sparse Row Format (CSR) on page 9
- Compressed Sparse Column Format (CSC) on page 10

Dense Format

The dense matrix `X` is represented by the following parameters (assuming it is stored in column-major format in memory):

- `m` (integer): the number of rows in the matrix.
- `n` (integer): the number of columns in the matrix.
- `ldX` (integer): the leading dimension of `X`, which must be greater than or equal to `m`. If `ldX` is greater than `m`, then `X` represents a sub-matrix of a larger $\texttt{idX} \times \texttt{n}$ 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 at least $\texttt{ldX} * \texttt{n}$ matrix elements and that CUSPARSE library functions may access values outside of the $\texttt{m} \times \texttt{n}$ sub-matrix, but will never overwrite them.

Figure 1 on page 8 shows a schematic representation of the $\texttt{m} \times \texttt{n}$ dense matrix `X` (shaded area) with leading dimension `ldX` greater than `m` and one-based indexing.

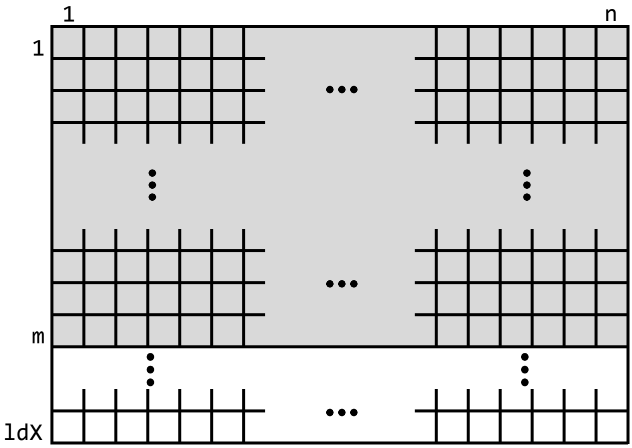

Figure 1. m × n Dense Matrix X with ldX > m

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

Coordinate Format (COO)

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

- `nnz` (integer): the number of non-zero elements in the matrix.
- `cooValA` (pointer): points to the data array of length `nnz` that holds all non-zero values of `A` in row-major format.
- `cooRowIndA` (pointer): points to the integer array of length `nnz` that contains the row indices of the corresponding elements in array `cooValA`.
- `cooColIndA` (pointer): points to the integer array of length `nnz` that contains the column indices of the corresponding elements in array `cooValA`.

Note: It is assumed that the indices are given in row-major format (first sorted by row indices and then within the same row by column indices) and that each pair of row and column indices appears only once.

Consider the following 4×5 matrix A.

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

This is how it is stored in COO zero-based format.

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

And this is the COO one-based format.

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

Compressed Sparse Row Format (CSR)

The only difference between the COO and CSR formats is that the array containing the row indices is compressed in CSR format.

The $m \times n$ sparse matrix `A` is represented in CSR format by the following parameters:

- `nnz` (integer): the number of non-zero elements in the matrix.
- `csrValA` (pointer): points to the data array of length `nnz` that holds all non-zero values of `A` in row-major format.
- `csrRowPtrA` (pointer): points to the integer array of length `m+1` that holds indices pointing to the array `csrColIndA/csrValA`. For the first `m` entries, `csrRowPtrA(i)` contains the index of the first non-zero element in the `i`th row, while the last entry, `csrRowPtrA(m)`, contains `nnz+csrRowPtrA(0)`. In general, `csrRowPtrA(0)` is `0` or `1` depending on whether zero- or one-based format is used, respectively.

- `csrColIndA` (pointer): points to the integer array of length `nnz` that holds the column indices of the corresponding elements in `csrValA`.

Note: It is assumed that the indices are given in row-major format (first sorted by row indices and then within the same row by column indices) and that each pair of row and column indices appears only once.

Again, consider the 4×5 matrix A.

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

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

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

This is the CSR one-based format.

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

Compressed Sparse Column Format (CSC)

The $m \times n$ matrix `A` is represented in CSC format by the following parameters:

- `nnz` (integer): the number of non-zero elements in the matrix.
- `cscValA` (pointer): points to the data array of length `nnz` that holds all non-zero values of `A` in column-major format.
- `cscRowIndA` (pointer): points to the integer array of length `nnz` that holds the row indices of the corresponding elements in `cscValA`.

- `cscColPtrA` (pointer): points to the integer array of length `n+1` that holds indices pointing to array `cscRowIndA/cscValA`. For the first `n` entries, `cscColPtrA(i)` contains the index of the first non-zero element in the `i`th column, while the last entry, `csrColPtrA(n)`, contains `nnz+csrColPtrA(0)`. In general, `cscColPtrA(0)` is `0` or `1` depending on whether zero- or one-based format is used, respectively.

Note: It is assumed that the indices are given in column-major format (first sorted by column indices and then within the same column by row indices) and that each pair of row and column indices appears only once.

Also note that matrix `A` in CSR format has exactly the same memory layout as its transpose in CSC format (and vice-versa).

Consider the 4×5 matrix A one more time.

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

The CSC zero-based storage format is below.

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

And, this is the CSC one-based format.

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

CUSPARSE Types

The library supports the following data types: `float`, `double`, `cuComplex`, and `cuDoubleComplex`. The first two are standard C data types, the second two are exported from `cuComplex.h`.

The CUSPARSE library provides these types:

cusparseHandle_t

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

cusparseMatrixType_t

This type indicates the type of matrix stored in sparse storage.

```
typedef enum {
    CUSPARSE_MATRIX_TYPE_GENERAL=0,
    CUSPARSE_MATRIX_TYPE_SYMMETRIC=1,
    CUSPARSE_MATRIX_TYPE_HERMITIAN=2,
    CUSPARSE_MATRIX_TYPE_TRIANGULAR=3
} cusparseMatrixType_t;
```

cusparseFillMode_t

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

```
typedef enum {
    CUSPARSE_FILL_MODE_LOWER=0,
    CUSPARSE_FILL_MODE_UPPER=1
} cusparseFillMode_t;
```

cusparseDiagType_t

This type indicates if the matrix diagonal entries are equal to one.

```
typedef enum {
    CUSPARSE_DIAG_TYPE_NON_UNIT=0,
    CUSPARSE_DIAG_TYPE_UNIT=1
} cusparseDiagType_t;
```

cusparseIndexBase_t

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

```
typedef enum {
    CUSPARSE_INDEX_BASE_ZERO=0,
    CUSPARSE_INDEX_BASE_ONE=1
} cusparseIndexBase_t;
```

cusparseMatDescr_t

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

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

cusparseOperation_t

Indicates which operations need to be performed with the sparse matrix.

```
typedef enum {
    CUSPARSE_OPERATION_NON_TRANSPOSE=0,
    CUSPARSE_OPERATION_TRANSPOSE=1,
    CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE=2
} cusparseOperation_t;
```

cusparseDirection_t

Indicates whether the elements of a matrix should be parsed by rows or by columns (regardless of row- or column-major storage format).

```
typedef enum {
    CUSPARSE_DIRECTION_ROW=0,
    CUSPARSE_DIRECTION_COLUMN=1
} cusparseDirection_t;
```

cusparseSolveAnalysisInfo_t

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

cusparseStatus_t

This is a status type returned by the library functions and can have the following defined values:

```
typedef enum{
    CUSPARSE_STATUS_SUCCESS=0,
    CUSPARSE_STATUS_NOT_INITIALIZED=1,
    CUSPARSE_STATUS_ALLOC_FAILED=2,
    CUSPARSE_STATUS_INVALID_VALUE=3,
    CUSPARSE_STATUS_ARCH_MISMATCH=4,
    CUSPARSE_STATUS_MAPPING_ERROR=5,
    CUSPARSE_STATUS_EXECUTION_FAILED=6,
    CUSPARSE_STATUS_INTERNAL_ERROR=7,
    CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED=8,
} cusparseStatus_t;
```

The status values are explained in the following table:

CUSPARSE Status Definitions

CUSPARSE_STATUS_SUCCESS

The operation completed successfully.

CUSPARSE_STATUS_NOT_INITIALIZED

The CUSPARSE library was not initialized. This is usually caused by the lack of a prior **cusparseCreate()** call, or by an error in CUDA or in the hardware setup.

To correct: call **cusparseCreate()** prior to the function call; and check that the hardware, an appropriate version of the driver, and the CUSPARSE library are correctly installed.

CUSPARSE_STATUS_ALLOC_FAILED

Resource allocation failed inside the CUSPARSE library. This is usually caused by a **cudaMalloc()** failure.

To correct: prior to the function call, deallocate previously allocated memory as much as possible.

CUSPARSE_STATUS_INVALID_VALUE

An unsupported value or parameter was passed to the function (a negative vector size, for example).

To correct: ensure that all the parameters being passed have valid values.

CUSPARSE Status Definitions (continued)

CUSPARSE_STATUS_ARCH_MISMATCH

Function requires a feature absent from the device architecture; usually caused by the lack of support for atomic operations or double precision.

To correct: compile and run the application on a device with appropriate compute capability, which is 1.1 for 32-bit atomic operations and 1.3 for double precision.

CUSPARSE_STATUS_MAPPING_ERROR

An access to GPU memory space failed, which is usually caused by a failure to bind a texture.

To correct: prior to the function call, unbind any previously bound textures.

CUSPARSE_STATUS_EXECUTION_FAILED

The GPU program failed to execute. This is often caused by a launch failure of the kernel on the GPU, which can be caused by multiple reasons.

To correct: check that the hardware, an appropriate version of the driver, and the CUSPARSE library are correctly installed.

CUSPARSE_STATUS_INTERNAL_ERROR

An internal CUSPARSE operation failed. This error is usually caused by a **cudaMemcpyAsync()** failure.

To correct: check that the hardware, an appropriate version of the driver, and the CUSPARSE library are correctly installed.

CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED

The matrix type is not supported by this function. This is usually caused by passing an invalid matrix descriptor to the function.

To correct: check that the fields in **cusparseMatDescr_t descrA** were set correctly.

CHAPTER

2

CUSPARSE Functions

This chapter discusses the CUSPARSE functions, which are divided into five groups, and the naming convention used for Sparse Level 1, Level 2, and Level 3 functions.

CUSPARSE Helper Functions

The CUSPARSE helper functions are as follows:

- cusparseCreate() on page 18
- cusparseDestroy() on page 19
- cusparseGetVersion() on page 20
- cusparseSetKernelStream() on page 20
- cusparseCreateMatDescr() on page 21
- cusparseDestroyMatDescr() on page 21
- cusparseSetMatType() on page 21
- cusparseGetMatType() on page 22
- cusparseSetMatFillMode() on page 22
- cusparseGetMatFillMode() on page 23
- cusparseSetMatDiagType() on page 23
- cusparseGetMatDiagType() on page 23
- cusparseSetMatIndexBase() on page 24
- cusparseGetMatIndexBase() on page 24
- cusparseCreateSolveAnalysisInfo() on page 25
- cusparseDestroySolveAnalysisInfo() on page 25

cusparseCreate()

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

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

Note: The CUSPARSE library requires hardware with at least 1.1 compute capability. Please see *NVIDIA CUDA C Programming Guide*, Appendix A for the list of all compute capabilities.

Output

`handle`	initialized pointer to a CUSPARSE context

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	CUSPARSE library initialized successfully
`CUSPARSE_STATUS_NOT_INITIALIZED`	if an error with CUDA or the hardware setup has been detected
`CUSPARSE_STATUS_ALLOC_FAILED`	
`CUSPARSE_STATUS_ARCH_MISMATCH`	if device compute capability is less than 1.1

i. See also CUSPARSE Status Definitions on page 15.

cusparseDestroy()

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

Releases CPU side resources used by the CUSPARSE library. The release of GPU side resources may be deferred until the application shuts down.

Input

`handle`	handle to a CUSPARSE context

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	CUSPARSE library shut down successfully

i. See also CUSPARSE Status Definitions on page 15.

cusparseGetVersion()

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

Returns the version number of the CUSPARSE library.

Input

`handle`	handle to a CUSPARSE context

Output

`version`	integer version number of the library

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	CUSPARSE library version was returned successfully
CUSPARSE_STATUS_NOT_INITIALIZED	

i. See also CUSPARSE Status Definitions on page 15.

cusparseSetKernelStream()

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

Sets the CUSPARSE stream in which the kernels will run.

Input

`handle`	handle to a CUSPARSE context

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	if stream provided has been set properly
CUSPARSE_STATUS_NOT_INITIALIZED	

i. See also CUSPARSE Status Definitions on page 15.

cusparseCreateMatDescr()

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

Initializes the **MatrixType** and **IndexBase** fields of the matrix descriptor to the default values **CUSPARSE_MATRIX_TYPE_GENERAL** and **CUSPARSE_INDEX_BASE_ZERO**, while leaving other fields uninitialized.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	matrix descriptor initialized successfully
CUSPARSE_STATUS_ALLOC_FAILED	if resources could not be allocated for the matrix descriptor

cusparseDestroyMatDescr()

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

Releases the memory allocated for the matrix descriptor.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	matrix descriptor destroyed successfully

cusparseSetMatType()

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

Sets the **MatrixType** field of the matrix descriptor `descrA`.

Input

`type`	one of the enumerated matrix types

Output

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	**`MatrixType`** field was set successfully
CUSPARSE_STATUS_INVALID_VALUE	if invalid value was passed in the `type` parameter

cusparseGetMatType()

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

Returns the **`MatrixType`** field of the matrix descriptor `descrA`.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

one of the enumerated matrix types

cusparseSetMatFillMode()

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

Sets the **`FillMode`** field of the matrix descriptor `descrA`.

Input

`fillMode`	one of the enumerated matrix types

Output

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	**`FillMode`** field was set successfully
CUSPARSE_STATUS_INVALID_VALUE	if invalid value was passed in the `fillMode` parameter

cusparseGetMatFillMode()

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

Returns the **FillMode** field of the matrix descriptor `descrA`.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

one of the enumerated fill-in types

cusparseSetMatDiagType()

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

Sets the **DiagType** field of the matrix descriptor `descrA`.

Input

`diagType`	one of the enumerated diagonal modes

Output

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	**DiagType** field was set successfully
CUSPARSE_STATUS_INVALID_VALUE	if invalid value was passed in the `diagType` parameter

cusparseGetMatDiagType()

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

Returns the **DiagType** field of the matrix descriptor `descrA`.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

one of the enumerated diagonal modes

cusparseSetMatIndexBase()

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

Sets the **IndexBase** field of the matrix descriptor `descrA`.

Input

`base`	one of the enumerated index base modes

Output

`descrA`	descriptor of the matrix `A`

Status Returned

CUSPARSE_STATUS_SUCCESS	**IndexBase** field was set successfully
CUSPARSE_STATUS_INVALID_VALUE	if invalid value was passed in the `base` parameter

cusparseGetMatIndexBase()

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

Returns the **IndexBase** field of the matrix descriptor `descrA`.

Input

`descrA`	descriptor of the matrix `A`

Status Returned

one of the enumerated index base modes

cusparseCreateSolveAnalysisInfo()

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

Creates and initializes the `info` structure to default values.

Input

`info`	`info` structure

Status Returned

CUSPARSE_STATUS_SUCCESS	if `info` structure was initialized successfully
CUSPARSE_STATUS_ALLOC_FAILED	if resources could not be allocated for the `info` structure

cusparseDestroySolveAnalysisInfo()

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

Destroys and releases any memory required by the info structure.

Input

`info`	`info` structure

Status Returned

CUSPARSE_STATUS_SUCCESS	if `info` structure was destroyed successfully

Naming Convention for the Sparse Level Functions

Most of the CUSPARSE 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>[<sparse data format>]<operation>[<sparse data format>]
```

with `<T>` being **`S`**, **`D`**, **`C`**, **`Z`**, or **`X`** corresponding to the data types `float`, `double`, **`cuComplex`**, **`cuDoubleComplex`**, or no type, respectively. All these functions have the return type **`cusparseStatus_t`**.

Sparse Level 1 Functions

This chapter describes the Level 1 sparse linear algebra functions that perform scalar- and vector-based operations. The following functions are implemented:

cusparse{S,D,C,Z}axpyi

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

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

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

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

Multiplies the vector x in sparse format by the constant alpha and adds the result to the vector y in dense format; that is, it overwrites y with $\mathrm{alpha} * \mathrm{x} + \mathrm{y}$.

For i = 0 to nnz-1

$$\mathrm{y}[\mathrm{xInd}[i] - \mathrm{idxBase}] = \mathrm{y}[\mathrm{xInd}[i] - \mathrm{idxBase}] + \mathrm{alpha} * \mathrm{xVal}[i]$$

Input

handle	handle to a CUSPARSE context
nnz	number of elements of the vector x
alpha	constant multiplier
xVal	nnz non-zero values of vector x
xInd	nnz indices corresponding to non-zero values of vector x

Input (continued)

`y`	initial vector in dense format
`idxBase`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**

Output

`y`	result (unchanged if `nnz == 0`)

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_INVALID_VALUE`	if `idxBase` is neither **`CUSPARSE_INDEX_BASE_ZERO`** nor **`CUSPARSE_INDEX_BASE_ONE`**
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}doti

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

```
cusparseStatus_t
cusparseDdoti(
    cusparseHandle_t handle, int nnz,
    const double *xVal, const int *xInd,
    const float *y, double *resultHostPtr,
    cusparseIndexBase_t idxBase )
```

```
cusparseStatus_t
cusparseCdoti(
    cusparseHandle_t handle, int nnz,
    const cuComplex *xVal, const int *xInd,
    const float *y, cuComplex *resultHostPtr,
    cusparseIndexBase_t idxBase )
```

```
cusparseStatus_t
cusparseZdoti(
    cusparseHandle_t handle, int nnz,
    const cuDoubleComplex *xVal, const int *xInd,
    const float *y, cuDoubleComplex *resultHostPtr,
    cusparseIndexBase_t idxBase )
```

Returns the dot product of a vector x in sparse format and vector y in dense format.

For i = 0 to nnz-1

$$\text{resultHostPtr} \mathrel{+}= \text{xVal}[i] * \text{y}[\text{xInd}[i - \text{idxBase}]]$$

Input

handle	handle to a CUSPARSE context
nnz	number of elements of the vector x
xVal	nnz non-zero values of vector x
xInd	nnz indices corresponding to non-zero values of vector x
y	vector in dense format
resultHostPtr	**pointer where to write the result in host memory**
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**

Output

resultHostPtr	**updated host memory with dot product (zero if** nnz == 0**)**

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_ALLOC_FAILED	
CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{C,Z}dotci

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

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

Returns the dot product of a complex conjugate of vector x in sparse format and complex vector y in dense format.

It computes the sum for i = 0 to nnz-1 of

$$\mathtt{resultHostPtr} \mathrel{+}= \overline{\mathtt{xVal}[i]} * \mathtt{y}[\mathtt{xInd}[i - \mathtt{idxBase}]]$$

Input

handle	handle to a CUSPARSE context
nnz	number of elements of the vector x
xVal	nnz non-zero values of vector x
xInd	nnz indices corresponding to non-zero values of vector x
y	vector in dense format
resultHostPtr	**pointer where to write the result in host memory**
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**

Output

resultHostPtr	**updated host memory with dot product (zero if** nnz == 0**)**

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_ALLOC_FAILED	
CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**

Status Returned[i] (continued)

CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}gthr

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

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

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

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

Gathers the elements of the vector `y` listed by the index array `xInd` into the array `xVal`.

Input

`handle`	handle to a CUSPARSE context
`nnz`	number of elements of the vector `x`
`y`	vector in dense format, of size greater than or equal to `max(xInd)` – `idxBase` + `1`

Input (continued)

xVal	pre-allocated array in device memory of size greater than or equal to nnz
xInd	nnz indices corresponding to non-zero values of vector x
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**

Output

xVal	updated vector of nnz elements (unchanged if nnz == 0)

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}gthrz

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

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

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

```
cusparseStatus_t
cusparseZgthrz(
    cusparseHandle_t handle, int nnz, cuDoubleComplex *y,
    cuDoubleComplex *xVal, const int *xInd,
    cusparseIndexBase_t idxBase )
```

Gathers the elements of the vector `y` listed by the index array `xInd` into the vector `x`, and zeroes those elements in the vector `y`.

Input

`handle`	handle to a CUSPARSE context
`nnz`	number of elements of the vector `x`
`y`	vector in dense format, of size greater than or equal to `max(xInd)` – `idxBase` + `1`
`xVal`	`nnz` non-zero values of vector `x`
`xInd`	`nnz` indices corresponding to non-zero values of vector `x`
`idxBase`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**

Output

`xVal`	`nnz` non-zero values of vector `x` (unchanged if `nnz == 0`)
`xInd`	`nnz` indices corresponding to non-zero values of vector `x` (unchanged if `nnz == 0`)
`y`	vector in dense format (unchanged if `nnz == 0`)

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_INVALID_VALUE`	if `idxBase` is neither **`CUSPARSE_INDEX_BASE_ZERO`** nor **`CUSPARSE_INDEX_BASE_ONE`**
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D}roti

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

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

Applies Givens rotation, defined by values `c` and `s`, to vectors `x` in sparse and `y` in dense format.

For i = `0` to `nnz-1`

$$\mathtt{y}[\mathtt{xInd}[i] - \mathtt{idxBase}] = \mathtt{c} * \mathtt{y}[\mathtt{xInd}[i] - \mathtt{idxBase}] - \mathtt{s} * \mathtt{xVal}[i]$$
$$\mathtt{x}[i] = \mathtt{c} * \mathtt{xVal}[i] + \mathtt{s} * \mathtt{y}[\mathtt{xInd}[i] - \mathtt{idxBase}]$$

Input

`handle`	handle to a CUSPARSE context
`nnz`	number of elements of the vector `x`
`xVal`	`nnz` non-zero values of vector `x`
`xInd`	`nnz` indices corresponding to non-zero values of vector `x`
`y`	vector in dense format
`c`	scalar
`s`	scalar
`idxBase`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**

Output

`xVal`	updated `nnz` non-zero values of vector `x` (unchanged if `nnz == 0`)
`xInd`	updated `nnz` indices corresponding to non-zero values of vector `x` (unchanged if `nnz == 0`)
`y`	updated vector in dense format (unchanged if `nnz == 0`)

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`
`CUSPARSE_STATUS_NOT_INITIALIZED`

Status Returned[i] (continued)

CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}sctr

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

Scatters 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.

Input

handle	handle to a CUSPARSE context
nnz	number of elements of the vector x

Input (continued)

`xVal`	`nnz` non-zero values of vector `x`
`xInd`	`nnz` indices corresponding to non-zero values of vector `x`
`y`	pre-allocated vector in dense format, of size greater than or equal to `max(xInd)` – `idxBase` + `1`
`idxBase`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**

Output

`y`	updated vector in dense format (unchanged if `nnz == 0`)

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_INVALID_VALUE`	if `idxBase` is neither **`CUSPARSE_INDEX_BASE_ZERO`** nor **`CUSPARSE_INDEX_BASE_ONE`**
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

Sparse Level 2 Functions

This chapter describes the Level 2 sparse linear algebra functions that perform matrix- and vector-based operations.

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

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

Finally, once all the solves have completed, the opaque data structure pointed to by the **`cusparseSolveAnalysisInfo_t`** parameter can be released by calling cusparseDestroySolveAnalysisInfo().

The following functions are implemented:

cusparse{S,D,C,Z}csrmv

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

Performs one of the matrix-vector operations

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

where $\mathrm{op}(A) = A$, $\mathrm{op}(A) = A^T$, or $\mathrm{op}(A) = A^H$.

`A` is an $m \times n$ matrix in CSR format, defined by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`; `alpha` and `beta` are scalars, and `x` and `y` are vectors in dense format.

Please note: when using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this function may produce different results for the same input parameters. In these cases, it uses atomic operations to compute the final result because many threads may be adding floating point numbers to the same memory location without any specific ordering. If exactly the same output is required for any input when multiplying by the transpose of a general matrix, the following procedure can be used.

1. Convert the matrix from CSR to CSC format using one of the **`csr2csc()`** functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.
2. Call the **`csrmv()`** function with the **`cusparseOperation_t`** parameter set to **`CUSPARSE_OPERATION_NON_TRANSPOSE`** and with the interchanged rows and columns of the matrix stored in CSC format. This (implicitly) multiplies the vector by the transpose of the matrix in the original CSR format.

Input

`handle`	handle to a CUSPARSE context
`transA`	specifies `op(A)`. See cusparseOperation_t on page 14.
`m`	specifies the number of rows of matrix `A`; `m` must be at least zero
`n`	specifies the number of columns of matrix `A`; `n` must be at least zero
`alpha`	scalar multiplier applied to $\mathrm{op}(A) * x$
`descrA`	descriptor of matrix `A`. Supported types are **`CUSPARSE_MATRIX_TYPE_GENERAL`**, **`CUSPARSE_MATRIX_TYPE_SYMMETRIC`**, and **`CUSPARSE_MATRIX_TYPE_HERMITIAN`**. Also, index bases **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`** are supported.

Input (continued)

csrValA	array of nnz elements, where nnz is the number of non-zero elements and can be obtained from csrRowPtrA(m) – csrRowPtrA(0)
csrRowPtrA	array of m+1 index elements
csrColIndA	array of nnz column indices
x	vector of n elements if op(A) = A, and m elements if op(A) = A^T or op(A) = A^H
beta	scalar multiplier applied to y. If beta is zero, y does not have to be a valid input.
y	vector of m elements if op(A) = A, and n elements if op(A) = A^T or op(A) = A^H

Output

y	updated according to $y = \text{alpha} * \text{op}(A) * x + \text{beta} * y$

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_ALLOC_FAILED	
CUSPARSE_STATUS_INVALID_VALUE	
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}csrsv_analysis

```
cusparseStatus_t
cusparseScsrsv_analysis(
    cusparseHandle_t handle, cusparseOperation_t transA,
    int m, const cusparseMatDescr_t descrA,
    const float *csrValA, const int *csrRowPtrA,
    const int *csrColIndA, cusparseSolveAnalysisInfo_t info )
```

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

Performs the analysis phase of the solution of a sparse triangular linear system

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

where $\mathrm{op}(A) = A$, $\mathrm{op}(A) = A^T$, or $\mathrm{op}(A) = A^H$.

It is expected to be executed only once for a given matrix and a particular operation type to be used in the solve phase. A is an $m \times n$ matrix in CSR format, defined by the three arrays csrValA, csrRowPtrA, and csrColIndA.

Input

handle	handle to a CUSPARSE context
transA	specifies op(A)
m	specifies the number of rows and columns of matrix A; m must be at least zero
descrA	descriptor of matrix A. The only types supported are matrix type **CUSPARSE_MATRIX_TYPE_TRIANGULAR** and diagonal type **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	array of nnz elements, where nnz is the number of non-zero elements and can be obtained from csrRowPtrA(m) – csrRowPtrA(0)
csrRowPtrA	array of m+1 index elements

Input (continued)

`csrColIndA`	array of `nnz` column indices
`info`	structure that stores the information collected during the analysis phase. It should be passed to the solve phase unchanged.

Output

`info`	structure that stores the information collected during the analysis phase. It should be passed to the solve phase unchanged.

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_ALLOC_FAILED	
CUSPARSE_STATUS_INVALID_VALUE	
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision (or does not have support for atomics).
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}csrsv_solve

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

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

Performs the solve phase of the solution of a sparse triangular linear system

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

where $\mathrm{op}(A) = A$, $\mathrm{op}(A) = A^T$, or $\mathrm{op}(A) = A^H$.

If needed, it can be executed multiple times for a given matrix and a particular operation type. `A` is an $m \times n$ matrix in CSR format, defined by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA`; `alpha` is a scalar, and `x` and `y` are vectors in dense format.

Input

`handle`	handle to a CUSPARSE context
`transA`	specifies `op(A)`
`m`	specifies the number of rows and columns of matrix `A`; `m` must be at least zero
`alpha`	scalar multiplier applied to `x`
`descrA`	descriptor of matrix `A`. The only types supported are matrix type **`CUSPARSE_MATRIX_TYPE_TRIANGULAR`** and diagonal type **`CUSPARSE_DIAG_TYPE_NON_UNIT`**.

Input (continued)

csrValA	array of nnz elements, where nnz is the number of non-zero elements and can be obtained from csrRowPtrA(m) – csrRowPtrA(0)
csrRowPtrA	array of m+1 index elements
csrColIndA	array of nnz column indices
info	structure that stores the information collected during the analysis phase. It should be passed to the solve phase unchanged.
x	vector of m elements
y	vector of m elements

Output

y	updated according to op(A) ∗ y = alpha ∗ x

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_INVALID_VALUE	
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision (or does not have support for atomics).
CUSPARSE_STATUS_MAPPING_ERROR	
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

Sparse Level 3 Function

This chapter describes the Level 3 sparse linear algebra function that performs operations involving sparse and tall dense matrices.

The following function is implemented:

cusparse{S,D,C,Z}csrmm

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

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

Performs one of these matrix-matrix operations:

$C = alpha * op(A) * B + beta * C$,

where $op(A) = A$, $op(A) = A^T$, or $op(A) = A^H$;

and alpha and beta are scalars. B and C are dense matrices stored in column-major format, A is an $m \times k$ matrix in CSR format, defined by the three arrays csrValA, csrRowPtrA and csrColIndA.

Please note: when using the (conjugate) transpose of a general matrix or a Hermitian/symmetric matrix, this function may produce different results for the same input parameters. In these cases, it uses atomic operations to compute the final result because many threads may be adding floating point numbers to the same memory location without any specific ordering. If exactly the same output is required for any input when multiplying by the transpose of a general matrix, the following procedure can be used.

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

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

Input

handle	handle to a CUSPARSE context
transA	specifies op(A). See cusparseOperation_t on page 14.
m	number of rows of matrix A; m must be at least zero.
n	number of columns of matrices B and C; n must be at least zero.
k	number of columns of matrix A; k must be at least zero.
alpha	scalar multiplier applied to $op(A) * B$
descrA	descriptor of matrix A. Supported types are **CUSPARSE_MATRIX_TYPE_GENERAL**, **CUSPARSE_MATRIX_TYPE_SYMMETRIC**, and **CUSPARSE_MATRIX_TYPE_HERMITIAN**. Also, index bases **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE** are supported.
csrValA	array of nnz elements, where nnz is the number of non-zero elements and can be obtained from $csrRowPtrA(m) - csrRowPtrA(0)$
csrRowPtrA	array of m+1 index elements
csrColIndA	array of nnz column indices
B	array of dimension (ldb, n)
ldb	leading dimension of B. It must be at least $max(1, k)$ if $op(A) = A$, and at least $max(1, m)$ if $op(A) = A^T$ or $op(A) = A^H$.
beta	scalar multiplier applied to C. If beta is zero, C does not have to be a valid input.
C	array of dimension (ldc, n)
ldc	leading dimension of C. It must be at least $max(1, m)$ if $op(A) = A$, and at least $max(1, k)$ if $op(A) = A^T$ or $op(A) = A^H$.

Output

C	updated according to $alpha * op(A) * B + beta * C$

Status Returned[i]

CUSPARSE_STATUS_SUCCESS

CUSPARSE_STATUS_NOT_INITIALIZED

CUSPARSE_STATUS_ALLOC_FAILED

CUSPARSE_STATUS_INVALID_VALUE

Status Returned[i] (continued)

CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

Format Conversion Functions

The format conversion functions are listed below:

cusparse{S,D,C,Z}nnz

```
cusparseStatus_t
cusparseSnnz(
    cusparseHandle_t handle, cusparseDirection_t dirA,
    int m, int n, const cusparseMatDescr_t descrA,
    const float *A, int lda, int *nnzPerVector,
    int *nnzHostPtr )
cusparseStatus_t
cusparseDnnz(
    cusparseHandle_t handle, cusparseDirection_t dirA,
    int m, int n, const cusparseMatDescr_t descrA,
    const double *A, int lda, int *nnzPerVector,
    int *nnzHostPtr )
cusparseStatus_t
cusparseCnnz(
    cusparseHandle_t handle, cusparseDirection_t dirA,
    int m, int n, const cusparseMatDescr_t descrA,
    const cuComplex *A, int lda, int *nnzPerVector,
    int *nnzHostPtr )
```

```
cusparseStatus_t
cusparseZnnz(
    cusparseHandle_t handle, cusparseDirection_t dirA,
    int m, int n, const cusparseMatDescr_t descrA,
    const cuDoubleComplex *A, int lda, int *nnzPerVector,
    int *nnzHostPtr )
```

Computes the number of non-zero elements per row or column and the total number of non-zero elements.

Input

`handle`	handle to a CUSPARSE context
`dirA`	**`CUSPARSE_DIRECTION_ROW`** or **`CUSPARSE_DIRECTION_COLUMN`** indicates whether to count the number of non-zero elements per row or per column, respectively.
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.
`descrA`	descriptor of matrix `A`. The only **`MatrixType`** supported is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. **`IndexBase`** constants **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`** are supported.
`A`	array of dimension `(lda, n)`
`lda`	leading dimension of `A`
`nnzPerVector`	array of size `m` or `n` containing the number of non-zero elements per row or column, respectively
`nnzHostPtr`	pointer (in the host memory) to an integer to be filled

Output

`nnzPerVector`	array of size `m` or `n` containing number of non-zero elements per row or column, respectively
`nnzHostPtr`	pointer (in the host memory) to an integer containing the total number of non-zero elements

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_ALLOC_FAILED`	
`CUSPARSE_STATUS_INVALID_VALUE`	
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.

Status Returned[i] (continued)

CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_INTERNAL_ERROR	
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}dense2csr

```
cusparseStatus_t
cusparseSdense2csr(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const float *A, int lda,
    const int *nnzPerRow, float *csrValA,
    int *csrRowPtrA, int *csrColIndA )
cusparseStatus_t
cusparseDdense2csr(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const double *A, int lda,
    const int *nnzPerRow, double *csrValA,
    int *csrRowPtrA, int *csrColIndA )
cusparseStatus_t
cusparseCdense2csr(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *A, int lda,
    const int *nnzPerRow, cuComplex *csrValA,
    int *csrRowPtrA, int *csrColIndA )
cusparseStatus_t
cusparseZdense2csr(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *A, int lda,
    const int *nnzPerRow, cuDoubleComplex *csrValA,
    int *csrRowPtrA, int *csrColIndA )
```

Converts the matrix `A` in dense format into a matrix in CSR format. All the parameters are pre-allocated by the user, and the arrays are filled in based on `nnzPerRow` (which can be pre-computed with **`cusparse{S,D,C,Z}nnz()`**).

Input

`handle`	handle to a CUSPARSE context
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.
`descrA`	descriptor of matrix `A`. The only **`MatrixType`** supported is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. **`IndexBase`** constants **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`** are supported.
`A`	array of dimension (`lda`, `n`)
`lda`	leading dimension of `A`
`nnzPerRow`	array of size `m` containing the number of non-zero elements per row
`csrValA`	array of `nnz` elements to be filled
`csrRowPtrA`	array of `m+1` index elements
`csrColIndA`	array of `nnz` column indices, corresponding to the non-zero elements in the matrix

Output

`csrValA`	updated array of `nnz` elements, where `nnz` is the number of non-zero elements in the matrix
`csrRowPtrA`	updated array of `m+1` index elements
`csrColIndA`	updated array of `nnz` column indices, corresponding to the non-zero elements in the matrix

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_ALLOC_FAILED`	
`CUSPARSE_STATUS_INVALID_VALUE`	
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU
`CUSPARSE_STATUS_INTERNAL_ERROR`	
`CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED`	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}csr2dense

```
cusparseStatus_t
cusparseScsr2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    float *A, int lda )
cusparseStatus_t
cusparseDcsr2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const double *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    double *A, int lda )
cusparseStatus_t
cusparseCcsr2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    cuComplex *A, int lda )
cusparseStatus_t
cusparseZcsr2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *csrValA,
    const int *csrRowPtrA, const int *csrColIndA,
    cuDoubleComplex *A, int lda )
```

Converts the matrix in CSR format defined by the three arrays `csrValA`, `csrRowPtrA`, and `csrColIndA` into a matrix `A` in dense format. The dense matrix `A` is filled in with the values of the sparse matrix and with zeros elsewhere.

Input

`handle`	handle to a CUSPARSE context
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.

Input (continued)

`descrA`	descriptor of matrix `A`. The only **`MatrixType`** supported is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. **`IndexBase`** constants **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`** are supported.
`csrValA`	array of `nnz` elements, where `nnz` is the number of non-zero elements and can be obtained from `csrRowPtrA(m)` – `csrRowPtrA(0)`
`csrRowPtrA`	array of `m+1` index elements
`csrColIndA`	array of `nnz` column indices
`A`	array of dimension (`lda`, `n`)
`lda`	leading dimension of `A`

Output

`A`	updated array filled in with values defined in the sparse matrix, and zeros elsewhere

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_INVALID_VALUE`	
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU
`CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED`	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}dense2csc

```
cusparseStatus_t
cusparseSdense2csc(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const float *A, int lda,
    const int *nnzPerCol, float *cscValA,
    int *cscRowIndA, int *cscColPtrA )

cusparseStatus_t
cusparseDdense2csc(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const double *A, int lda,
    const int *nnzPerCol, double *cscValA,
    int *cscRowIndA, int *cscColPtrA )

cusparseStatus_t
cusparseCdense2csc(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *A, int lda,
    const int *nnzPerCol, cuComplex *cscValA,
    int *cscRowIndA, int *cscColPtrA )

cusparseStatus_t
cusparseZdense2csc(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *A, int lda,
    const int *nnzPerCol, cuDoubleComplex *cscValA,
    int *cscRowIndA, int *cscColPtrA )
```

Converts the matrix `A` in dense format into a matrix in CSC format. All the parameters are pre-allocated by the user, and the arrays are filled in based on `nnzPerCol` (which can be pre-computed with **`cusparse{S,D,C,Z}nnz()`**).

Input

`handle`	handle to a CUSPARSE context
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.

Input (continued)

descrA	descriptor of matrix A. The only **MatrixType** supported is **CUSPARSE_MATRIX_TYPE_GENERAL**. **IndexBase** constants **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE** are supported.
A	array of dimension (lda, n)
lda	leading dimension of A
nnzPerCol	array of size n containing the number of non-zero elements per column
cscValA	array of nnz elements to be filled
cscRowIndA	array of nnz row indices, corresponding to the non-zero elements in the matrix
cscColPtrA	array with n+1 index elements

Output

cscValA	updated array of nnz elements, where nnz is the number of non-zero elements in the matrix
cscRowIndA	updated array of nnz row indices, corresponding to the non-zero elements in the matrix
cscColPtrA	updated array with n+1 index elements

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision.
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}csc2dense

```
cusparseStatus_t
cusparseScsc2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const float *cscValA,
    const int *cscRowIndA, const int *cscColPtrA,
    float *A, int lda )
cusparseStatus_t
cusparseDcsc2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const double *cscValA,
    const int *cscRowIndA, const int *cscColPtrA,
    double *A, int lda )
cusparseStatus_t
cusparseCcsc2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuComplex *cscValA,
    const int *cscRowIndA, const int *cscColPtrA,
    cuComplex *A, int lda )
cusparseStatus_t
cusparseZcsc2dense(
    cusparseHandle_t handle, int m, int n,
    const cusparseMatDescr_t descrA,
    const cuDoubleComplex *cscValA,
    const int *cscRowIndA, const int *cscColPtrA,
    cuDoubleComplex *A, int lda )
```

Converts the matrix in CSC format defined by the three arrays `cscValA`, `cscColPtrA`, and `cscRowIndA` into matrix `A` in dense format. The dense matrix `A` is filled in with the values of the sparse matrix and with zeros elsewhere.

Input

`handle`	handle to a CUSPARSE context
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.

Input (continued)

descrA	descriptor of matrix A. The only **MatrixType** supported is **CUSPARSE_MATRIX_TYPE_GENERAL**. **IndexBase** constants **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE** are supported.
cscValA	array of nnz elements, where nnz is the number of non-zero elements and can be obtained from cscColPtrA(m) – cscColPtrA(0)
cscRowIndA	array of nnz row indices
cscColPtrA	array of n+1 index elements
A	array of dimension (lda, n)
lda	leading dimension of A

Output

A	updated array filled in with values defined in the sparse matrix and zeros elsewhere

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_INVALID_VALUE	
CUSPARSE_STATUS_ARCH_MISMATCH	if the **D** or **Z** variants of the function were invoked on a device that does not support double precision
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	

i. See also CUSPARSE Status Definitions on page 15.

cusparse{S,D,C,Z}csr2csc

```
cusparseStatus_t
cusparseScsr2csc(
    cusparseHandle_t handle, int m, int n,
    const float *csrVal, const int *csrRowPtr,
    const int *csrColInd, float *cscVal,
    int *cscRowInd, int *cscColPtr,
    int copyValues, int base )
```

```
cusparseStatus_t
cusparseDcsr2csc(
    cusparseHandle_t handle, int m, int n,
    const double *csrVal,
    const int *csrRowPtr,
    const int *csrColInd, double *cscVal,
    int *cscRowInd, int *cscColPtr,
    int copyValues, int base )
cusparseStatus_t
cusparseCcsr2csc(
    cusparseHandle_t handle, int m, int n,
    const cuComplex *csrVal, const int *csrRowPtr,
    const int *csrColInd, cuComplex *cscVal,
    int *cscRowInd, int *cscColPtr,
    int copyValues, int base )
cusparseStatus_t
cusparseZcsr2csc(
    cusparseHandle_t handle, int m, int n,
    const cuDoubleComplex *csrVal, const int *csrRowPtr,
    const int *csrColInd, cuDoubleComplex *cscVal,
    int *cscRowInd, int *cscColPtr,
    int copyValues, int base )
```

Converts the matrix in CSR format defined with the three arrays `csrVal`, `csrRowPtr`, and `csrColInd` into matrix `A` in CSC format defined by arrays `cscVal`, `cscRowInd`, and `cscColPtr`. The resulting matrix can also be seen as the transpose of the original sparse matrix. This routine can also be used to convert a matrix in CSC format into a matrix in CSR format.

Input

`handle`	handle to a CUSPARSE context
`m`	number of rows of the matrix `A`; `m` must be at least zero.
`n`	number of columns of matrix `A`; `n` must be at least zero.
`descrA`	descriptor of matrix `A`. The only **`MatrixType`** supported is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. **`IndexBase`** constants **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`** are supported.
`csrVal`	array of `nnz` elements, where `nnz` is the number of non-zero elements and can be obtained from `csrRowPtrA(m)` – `csrRowPtrA(0)`

Input (continued)

`csrRowPtr`	array of `m+1` indices
`csrColInd`	array of `nnz` column indices
`cscVal`	array of `nnz` elements, where `nnz` is the number of non-zero elements and can be obtained from `csrColPtr(m)` – `csrColPtr(0)`
`cscRowInd`	array of `nnz` row indices
`cscColPtr`	array of `n+1` indices
`copyValues`	if zero, `cscVal` array is not filled
`base`	base index: **`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**

Output

`cscVal`	if `copyValues` is non-zero, updated array
`cscColPtr`	updated array of `n+1` index elements
`cscRowInd`	updated array of `nnz` row indices,

Status Returned[i]

`CUSPARSE_STATUS_SUCCESS`	
`CUSPARSE_STATUS_NOT_INITIALIZED`	
`CUSPARSE_STATUS_ALLOC_FAILED`	
`CUSPARSE_STATUS_INVALID_VALUE`	
`CUSPARSE_STATUS_ARCH_MISMATCH`	if the **`D`** or **`Z`** variants of the function were invoked on a device that does not support double precision.
`CUSPARSE_STATUS_EXECUTION_FAILED`	function failed to launch on GPU
`CUSPARSE_STATUS_INTERNAL_ERROR`	

i. See also CUSPARSE Status Definitions on page 15.

cusparseXcoo2csr

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

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).

Input

handle	handle to a CUSPARSE context
cooRowInd	array of row indices
nnz	number of non-zeros of the matrix in COO format; this is also the length of array cooRowInd
m	number of rows of the matrix A; m must be at least zero.
csrRowPtr	array of row pointers
idxBase	base index: **CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**

Output

csrRowPtr	updated array of m+1 index elements

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

cusparseXcsr2coo

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

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 pointers (corresponding to CSC format) into an array of uncompressed column indices (corresponding to COO format).

Input

handle	handle to a CUSPARSE context
csrRowPtr	array of compressed row pointers
nnz	number of non-zeros of the matrix in COO format; this is also the length of array cooRowInd
m	number of rows of the matrix A; m must be at least zero.
cooRowInd	array of uncompressed row indices
idxBase	base index: **CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**

Output

cooRowInd	updated array of nnz index elements

Status Returned[i]

CUSPARSE_STATUS_SUCCESS	
CUSPARSE_STATUS_NOT_INITIALIZED	
CUSPARSE_STATUS_INVALID_VALUE	if idxBase is neither **CUSPARSE_INDEX_BASE_ZERO** nor **CUSPARSE_INDEX_BASE_ONE**
CUSPARSE_STATUS_EXECUTION_FAILED	function failed to launch on GPU

i. See also CUSPARSE Status Definitions on page 15.

APPENDIX

A

CUSPARSE Library Example

Example A.1 on page 64 demonstrates an application of the CUSPARSE library. The example performs these actions:

1. Creates a sparse test matrix in COO format.
2. Creates a sparse and dense vector.
3. Allocates GPU memory and copies the matrix and vectors into it.
4. Initializes the CUSPARSE library.
5. Creates and sets up the matrix descriptor.
6. Converts the matrix from COO to CSR format.
7. Exercises Level 1 routines.
8. Exercises Level 2 routines.
9. Exercises Level 3 routines.

Example A.1. CUSPARSE Library Example

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

#define CLEANUP(s)                                   \
do {                                                 \
    printf ("%s\n", s);                              \
    if (yHostPtr)           free(yHostPtr);          \
    if (zHostPtr)           free(zHostPtr);          \
    if (xIndHostPtr)        free(xIndHostPtr);       \
    if (xValHostPtr)        free(xValHostPtr);       \
    if (cooRowIndexHostPtr) free(cooRowIndexHostPtr);\
    if (cooColIndexHostPtr) free(cooColIndexHostPtr);\
    if (cooValHostPtr)      free(cooValHostPtr);     \
    if (y)                  cudaFree(y);             \
    if (z)                  cudaFree(z);             \
    if (xInd)               cudaFree(xInd);          \
    if (xVal)               cudaFree(xVal);          \
    if (csrRowPtr)          cudaFree(csrRowPtr);     \
    if (cooRowIndex)        cudaFree(cooRowIndex);   \
    if (cooColIndex)        cudaFree(cooColIndex);   \
    if (cooVal)             cudaFree(cooVal);        \
    if (handle)             cusparseDestroy(handle); \
    fflush (stdout);                                 \
} while (0)

int main(){
    cudaError_t cudaStat1,cudaStat2,cudaStat3,cudaStat4,cudaStat5,cudaStat6;
    cusparseStatus_t status;
    cusparseHandle_t handle=0;
    cusparseMatDescr_t descra=0;
    int *    cooRowIndexHostPtr=0;
    int *    cooColIndexHostPtr=0;
    double * cooValHostPtr=0;
    int *    cooRowIndex=0;
```

Example A.1. CUSPARSE Library Example (continued)

```
    int *    cooColIndex=0;
    double * cooVal=0;
    int *    xIndHostPtr=0;
    double * xValHostPtr=0;
    double * yHostPtr=0;
    int *    xInd=0;
    double * xVal=0;
    double * y=0;
    int *    csrRowPtr=0;
    double * zHostPtr=0;
    double * z=0;
    int      n, nnz, nnz_vector, i, j;


    printf("testing example\n");
    /* create the following sparse test matrix in COO format */
    /* |1.0     2.0 3.0|
       |    4.0        |
       |5.0     6.0 7.0|
       |    8.0     9.0| */
    n=4; nnz=9;
    cooRowIndexHostPtr = (int *)   malloc(nnz*sizeof(cooRowIndexHostPtr[0]));
    cooColIndexHostPtr = (int *)   malloc(nnz*sizeof(cooColIndexHostPtr[0]));
    cooValHostPtr      = (double *)malloc(nnz*sizeof(cooValHostPtr[0]));
    if ((!cooRowIndexHostPtr) || (!cooColIndexHostPtr) || (!cooValHostPtr)){
        CLEANUP("Host malloc failed (matrix)");
        return EXIT_FAILURE;
    }
    cooRowIndexHostPtr[0]=0; cooColIndexHostPtr[0]=0; cooValHostPtr[0]=1.0;
    cooRowIndexHostPtr[1]=0; cooColIndexHostPtr[1]=2; cooValHostPtr[1]=2.0;
    cooRowIndexHostPtr[2]=0; cooColIndexHostPtr[2]=3; cooValHostPtr[2]=3.0;
    cooRowIndexHostPtr[3]=1; cooColIndexHostPtr[3]=1; cooValHostPtr[3]=4.0;
    cooRowIndexHostPtr[4]=2; cooColIndexHostPtr[4]=0; cooValHostPtr[4]=5.0;
    cooRowIndexHostPtr[5]=2; cooColIndexHostPtr[5]=2; cooValHostPtr[5]=6.0;
    cooRowIndexHostPtr[6]=2; cooColIndexHostPtr[6]=3; cooValHostPtr[6]=7.0;
    cooRowIndexHostPtr[7]=3; cooColIndexHostPtr[7]=1; cooValHostPtr[7]=8.0;
    cooRowIndexHostPtr[8]=3; cooColIndexHostPtr[8]=3; cooValHostPtr[8]=9.0;
```

Example A.1. CUSPARSE Library Example (continued)

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

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

Example A.1. CUSPARSE Library Example (continued)

```
        printf("xValHostPtr[%d]=%f\n",i,xValHostPtr[i]);
    }

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

Example A.1. CUSPARSE Library Example (continued)

```
    if ((cudaStat1 != cudaSuccess) ||
        (cudaStat2 != cudaSuccess) ||
        (cudaStat3 != cudaSuccess) ||
        (cudaStat4 != cudaSuccess) ||
        (cudaStat5 != cudaSuccess) ||
        (cudaStat6 != cudaSuccess)) {
        CLEANUP("Memcpy from Host to Device failed");
        return EXIT_FAILURE;
    }


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

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

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

Example A.1. CUSPARSE Library Example (continued)

```
        return EXIT_FAILURE;
    }
    //csrRowPtr = [0 3 4 7 9]

    /* exercise Level 1 routines (scatter vector elements) */
    status= cusparseDsctr(handle, nnz_vector, xVal, xInd,
                          &y[n], CUSPARSE_INDEX_BASE_ZERO);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Scatter from sparse to dense vector failed");
        return EXIT_FAILURE;
    }
    //y = [10 20 30 40 | 100 200 70 400]

    /* exercise Level 2 routines (csrmv) */
    status= cusparseDcsrmv(handle,CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, 2.0,
                           descra, cooVal, csrRowPtr, cooColIndex, &y[0],
                           3.0, &y[n]);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix-vector multiplication failed");
        return EXIT_FAILURE;
    }

    /* print intermediate results (y) */
    //y = [10 20 30 40 | 680 760 1230 2240]
    cudaMemcpy(yHostPtr, y, (size_t)(2*n*sizeof(y[0])), cudaMemcpyDeviceToHost);
    printf("Intermediate results:\n");
    for (j=0; j<2; j++){
        for (i=0; i<n; i++){
            printf("yHostPtr[%d,%d]=%f\n",i,j,yHostPtr[i+n*j]);
        }
    }

    /* exercise Level 3 routines (csrmm) */
    cudaStat1 = cudaMalloc((void**)&z, 2*(n+1)*sizeof(z[0]));
    if (cudaStat1 != cudaSuccess) {
        CLEANUP("Device malloc failed (z)");
        return EXIT_FAILURE;
```

Example A.1. CUSPARSE Library Example (continued)

```
    }
    cudaStat1 = cudaMemset((void *)z,0, 2*(n+1)*sizeof(z[0]));
    if (cudaStat1 != cudaSuccess) {
        CLEANUP("Memset on Device failed");
        return EXIT_FAILURE;
    }
    status= cusparseDcsrmm(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, 2, n,
                           5.0, descra, cooVal, csrRowPtr, cooColIndex, y, n,
                           0.0, z, n+1);
    if (status != CUSPARSE_STATUS_SUCCESS) {
        CLEANUP("Matrix-matrix multiplication failed");
        return EXIT_FAILURE;
    }

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

    /* check the results */
    /* Notice that CLEANUP() contains a call to cusparseDestroy(handle) */
    if ((zHostPtr[0] != 950.0)    ||
        (zHostPtr[1] != 400.0)    ||
        (zHostPtr[2] != 2550.0)   ||
        (zHostPtr[3] != 2600.0)   ||
        (zHostPtr[4] != 0.0)      ||
```

Example A.1. CUSPARSE Library Example (continued)

```
        (zHostPtr[5] != 49300.0)  ||
        (zHostPtr[6] != 15200.0)  ||
        (zHostPtr[7] != 132300.0) ||
        (zHostPtr[8] != 131200.0) ||
        (zHostPtr[9] != 0.0)      ||
        (yHostPtr[0] != 10.0)     ||
        (yHostPtr[1] != 20.0)     ||
        (yHostPtr[2] != 30.0)     ||
        (yHostPtr[3] != 40.0)     ||
        (yHostPtr[4] != 680.0)    ||
        (yHostPtr[5] != 760.0)    ||
        (yHostPtr[6] != 1230.0)   ||
        (yHostPtr[7] != 2240.0)){
        CLEANUP("example test FAILED");
        return EXIT_FAILURE;
    }
    else{
        CLEANUP("example test PASSED");
        return EXIT_SUCCESS;
    }
}
```

APPENDIX

B

CUSPARSE Fortran Bindings

CUSPARSE 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.1., "Fortran 77 CUSPARSE Library Example" on page 75.

Example B.1 shows Example A.1 on page 64 implemented in Fortran 77 on the host. This example should be compiled with `ARCH_64` defined as `1` on a 64-bit OS system and as undefined on a 32-bit OS system. For example, on g95 or gfortran, it can be done directly on the command line using the option `'-cpp -DARCH_64=1'`.

Example B.1. Fortran 77 CUSPARSE Library Example

```
program cusparse_fortran_example
      implicit none
      integer cuda_malloc
      external cuda_free
      integer cuda_memcpy_c2fort_int
      integer cuda_memcpy_c2fort_real
      integer cuda_memcpy_fort2c_int
      integer cuda_memcpy_fort2c_real
      integer cuda_memset
      integer cusparse_create
      external cusparse_destroy
      integer cusparse_get_version
      integer cusparse_create_mat_descr
      external cusparse_destroy_mat_descr
      integer cusparse_set_mat_type
      integer cusparse_get_mat_type
      integer cusparse_get_mat_fill_mode
      integer cusparse_get_mat_diag_type
      integer cusparse_set_mat_index_base
      integer cusparse_get_mat_index_base
      integer cusparse_xcoo2csr
      integer cusparse_dsctr
      integer cusparse_dcsrmv
      integer cusparse_dcsrmm
      external get_shifted_address
#if ARCH_64
      integer*8 handle
      integer*8 descrA
      integer*8 cooRowIndex
      integer*8 cooColIndex
      integer*8 cooVal
      integer*8 xInd
      integer*8 xVal
      integer*8 y
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
      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
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
      write(*,*) "testing fortran example"
c     predefined constants (need to be careful with them)
      dzero = 0.0
      dtwo  = 2.0
      dthree= 3.0
      dfive = 5.0
c     create the following sparse test matrix in COO format
c     (notice one-based indexing)
c     |1.0     2.0 3.0|
c     |    4.0        |
c     |5.0     6.0 7.0|
c     |    8.0     9.0|
      cooRowIndexHostPtr(1)=1
      cooColIndexHostPtr(1)=1
      cooValHostPtr(1)     =1.0
      cooRowIndexHostPtr(2)=1
      cooColIndexHostPtr(2)=3
      cooValHostPtr(2)     =2.0
      cooRowIndexHostPtr(3)=1
      cooColIndexHostPtr(3)=4
      cooValHostPtr(3)     =3.0
      cooRowIndexHostPtr(4)=2
      cooColIndexHostPtr(4)=2
      cooValHostPtr(4)     =4.0
      cooRowIndexHostPtr(5)=3
      cooColIndexHostPtr(5)=1
      cooValHostPtr(5)     =5.0
      cooRowIndexHostPtr(6)=3
      cooColIndexHostPtr(6)=3
      cooValHostPtr(6)     =6.0
      cooRowIndexHostPtr(7)=3
      cooColIndexHostPtr(7)=4
      cooValHostPtr(7)     =7.0
      cooRowIndexHostPtr(8)=4
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

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

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

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
         do i=1,n
            write(*,*) "yHostPtr[",i,",",j,"]=",yHostPtr(i+n*(j-1))
         enddo
      enddo
      do i=1,nnz_vector
         write(*,*) "xIndHostPtr[",i,"]=",xIndHostPtr(i)
         write(*,*) "xValHostPtr[",i,"]=",xValHostPtr(i)
      enddo

c     allocate GPU memory and copy the matrix and vectors into it
c     cudaSuccess=0
c     cudaMemcpyHostToDevice=1
      cudaStat1 = cuda_malloc(cooRowIndex,nnz*4)
      cudaStat2 = cuda_malloc(cooColIndex,nnz*4)
      cudaStat3 = cuda_malloc(cooVal,     nnz*8)
      cudaStat4 = cuda_malloc(y,          2*n*8)
      cudaStat5 = cuda_malloc(xInd,nnz_vector*4)
      cudaStat6 = cuda_malloc(xVal,nnz_vector*8)
      if ((cudaStat1 /= 0) .OR.
     $    (cudaStat2 /= 0) .OR.
     $    (cudaStat3 /= 0) .OR.
     $    (cudaStat4 /= 0) .OR.
     $    (cudaStat5 /= 0) .OR.
     $    (cudaStat6 /= 0)) then
         write(*,*) "Device malloc failed"
         write(*,*) "cudaStat1=",cudaStat1
         write(*,*) "cudaStat2=",cudaStat2
         write(*,*) "cudaStat3=",cudaStat3
         write(*,*) "cudaStat4=",cudaStat4
         write(*,*) "cudaStat5=",cudaStat5
         write(*,*) "cudaStat6=",cudaStat6
         stop
      endif
      cudaStat1 = cuda_memcpy_fort2c_int(cooRowIndex,cooRowIndexHostPtr,
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

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

c     initialize cusparse library
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

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

c      create and setup the matrix descriptor
c      CUSPARSE_STATUS_SUCCESS=0
c      CUSPARSE_MATRIX_TYPE_GENERAL=0
c      CUSPARSE_INDEX_BASE_ONE=1
       status= cusparse_create_mat_descr(descrA)
       if (status /= 0) then
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

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

c     exercise conversion routines: convert matrix from COO 2 CSR format
c     cudaSuccess=0
c     CUSPARSE_STATUS_SUCCESS=0
c     CUSPARSE_INDEX_BASE_ONE=1
      cudaStat1 = cuda_malloc(csrRowPtr,(n+1)*4)
      if (cudaStat1 /= 0) then
         call cuda_free(cooRowIndex)
         call cuda_free(cooColIndex)
         call cuda_free(cooVal)
         call cuda_free(xInd)
         call cuda_free(xVal)
         call cuda_free(y)
         call cusparse_destroy_mat_descr(descrA)
         call cusparse_destroy(handle)
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
            write(*,*) "Device malloc failed (csrRowPtr)"
            stop
         endif
         status= cusparse_xcoo2csr(handle,cooRowIndex,nnz,n,
     $                             csrRowPtr,1)
         if (status /= 0) then
            call cuda_free(cooRowIndex)
            call cuda_free(cooColIndex)
            call cuda_free(cooVal)
            call cuda_free(xInd)
            call cuda_free(xVal)
            call cuda_free(y)
            call cuda_free(csrRowPtr)
            call cusparse_destroy_mat_descr(descrA)
            call cusparse_destroy(handle)
            write(*,*) "Conversion from COO to CSR format failed"
            stop
         endif
c        csrRowPtr = [0 3 4 7 9]

c        exercise Level 1 routines (scatter vector elements)
c        CUSPARSE_STATUS_SUCCESS=0
c        CUSPARSE_INDEX_BASE_ONE=1
         call get_shifted_address(y,n*8,ynp1)
         status= cusparse_dsctr(handle, nnz_vector, xVal, xInd,
     $                          ynp1, 1)
         if (status /= 0) then
            call cuda_free(cooRowIndex)
            call cuda_free(cooColIndex)
            call cuda_free(cooVal)
            call cuda_free(xInd)
            call cuda_free(xVal)
            call cuda_free(y)
            call cuda_free(csrRowPtr)
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
            call cusparse_destroy_mat_descr(descrA)
            call cusparse_destroy(handle)
            write(*,*) "Scatter from sparse to dense vector failed"
            stop
         endif
c        y = [10 20 30 40 | 100 200 70 400]

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

c        print intermediate results (y)
c        y = [10 20 30 40 | 680 760 1230 2240]
c        cudaSuccess=0
c        cudaMemcpyDeviceToHost=2
         cudaStat1 = cuda_memcpy_c2fort_real(yHostPtr, y, 2*n*8, 2)
         if (cudaStat1 /= 0) then
            call cuda_free(cooRowIndex)
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
         call cuda_free(cooColIndex)
         call cuda_free(cooVal)
         call cuda_free(xInd)
         call cuda_free(xVal)
         call cuda_free(y)
         call cuda_free(csrRowPtr)
         call cusparse_destroy_mat_descr(descrA)
         call cusparse_destroy(handle)
         write(*,*) "Memcpy from Device to Host failed"
         stop
      endif
      write(*,*) "Intermediate results:"
      do j=1,2
         do i=1,n
             write(*,*) "yHostPtr[",i,",",j,"]=",yHostPtr(i+n*(j-1))
         enddo
      enddo

c     exercise Level 3 routines (csrmm)
c     cudaSuccess=0
c     CUSPARSE_STATUS_SUCCESS=0
c     CUSPARSE_OPERATION_NON_TRANSPOSE=0
      cudaStat1 = cuda_malloc(z, 2*(n+1)*8)
      if (cudaStat1 /= 0) then
         call cuda_free(cooRowIndex)
         call cuda_free(cooColIndex)
         call cuda_free(cooVal)
         call cuda_free(xInd)
         call cuda_free(xVal)
         call cuda_free(y)
         call cuda_free(csrRowPtr)
         call cusparse_destroy_mat_descr(descrA)
         call cusparse_destroy(handle)
         write(*,*) "Device malloc failed (z)"
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
         stop
      endif
      cudaStat1 = cuda_memset(z, 0, 2*(n+1)*8)
      if (cudaStat1 /= 0) then
         call cuda_free(cooRowIndex)
         call cuda_free(cooColIndex)
         call cuda_free(cooVal)
         call cuda_free(xInd)
         call cuda_free(xVal)
         call cuda_free(y)
         call cuda_free(z)
         call cuda_free(csrRowPtr)
         call cusparse_destroy_mat_descr(descrA)
         call cusparse_destroy(handle)
         write(*,*) "Memset on Device failed"
         stop
      endif
      status= cusparse_dcsrmm(handle, 0, n, 2, n, dfive,
     $                        descrA, cooVal, csrRowPtr, cooColIndex,
     $                        y, n, dzero, z, n+1)
      if (status /= 0) then
         call cuda_free(cooRowIndex)
         call cuda_free(cooColIndex)
         call cuda_free(cooVal)
         call cuda_free(xInd)
         call cuda_free(xVal)
         call cuda_free(y)
         call cuda_free(z)
         call cuda_free(csrRowPtr)
         call cusparse_destroy_mat_descr(descrA)
         call cusparse_destroy(handle)
         write(*,*) "Matrix-matrix multiplication failed"
         stop
      endif
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

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

c     check the results
      epsilon = 0.00000000000001
      if ((DABS(zHostPtr(1) - 950.0)   .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(2) - 400.0)   .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(3) - 2550.0)  .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(4) - 2600.0)  .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(5) - 0.0)     .GT. epsilon)  .OR.
```

Example B.1. Fortran 77 CUSPARSE Library Example (continued)

```
     $    (DABS(zHostPtr(6) - 49300.0) .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(7) - 15200.0) .GT. epsilon)  .OR.
     $    (DABS(zHostPtr(8) - 132300.0).GT. epsilon)  .OR.
     $    (DABS(zHostPtr(9) - 131200.0).GT. epsilon)  .OR.
     $    (DABS(zHostPtr(10) - 0.0)    .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(1) - 10.0)    .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(2) - 20.0)    .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(3) - 30.0)    .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(4) - 40.0)    .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(5) - 680.0)   .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(6) - 760.0)   .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(7) - 1230.0)  .GT. epsilon)  .OR.
     $    (DABS(yHostPtr(8) - 2240.0)  .GT. epsilon)) then
          write(*,*) "fortran example test FAILED"
       else
          write(*,*) "fortran example test PASSED"
       endif

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

       stop
       end
```