hipSPARSE Documentation

Advanced Micro Devices

Sep 14, 2022

CONTENTS:

hipSPARSE exposes a common interface that provides Basic Linear Algebra Subroutines for sparse computation implemented on top of AMD's Radeon Open Compute ROCm runtime and toolchains. hipSPARSE is created using the HIP programming language and optimized for AMD's latest discrete GPUs.

In the following, three separate chapters are available:

- *User Manual*: This is the manual of hipSPARSE. It can be seen as a starting guide for new users but also a reference book for more experienced developers.
- *API*: This is a list of API functions provided by hipSPARSE.

CHAPTER

ONE

USER MANUAL

1.1 Introduction

hipSPARSE is a library that contains basic linear algebra subroutines for sparse matrices and vectors written in HIP for GPU devices. It is designed to be used from C and C++ code. The functionality of hipSPARSE is organized in the following categories:

- *Sparse Auxiliary Functions* describe available helper functions that are required for subsequent library calls.
- *Sparse Level 1 Functions* describe operations between a vector in sparse format and a vector in dense format.
- *Sparse Level 2 Functions* describe operations between a matrix in sparse format and a vector in dense format.
- *Sparse Level 3 Functions* describe operations between a matrix in sparse format and multiple vectors in dense format.
- *Sparse Extra Functions* describe operations that manipulate sparse matrices.
- *Preconditioner Functions* describe manipulations on a matrix in sparse format to obtain a preconditioner.
- *Sparse Conversion Functions* describe operations on a matrix in sparse format to obtain a different matrix format.
- *Sparse Reordering Functions* describe operations on a matrix in sparse format to obtain a reordering.
- *Sparse Generic Functions* describe operations that manipulate sparse matrices.

The code is open and hosted here: https://github.com/ROCmSoftwarePlatform/hipSPARSE

hipSPARSE is a SPARSE marshalling library, with multiple supported backends. It sits between the application and a *worker* SPARSE library, marshalling inputs into the backend library and marshalling results back to the application. hipSPARSE exports an interface that does not require the client to change, regardless of the chosen backend. Currently, hipSPARSE supports rocSPARSE and cuSPARSE as backends. hipSPARSE focuses on convenience and portability. If performance outweighs these factors, then using rocSPARSE itself is highly recommended. rocSPARSE can also be found on GitHub.

1.2 Building and Installing

1.2.1 Prerequisites

hipSPARSE requires a ROCm enabled platform, more information here.

1.2.2 Installing pre-built packages

hipSPARSE can be installed from AMD ROCm repository. For detailed instructions on how to set up ROCm on different platforms, see the AMD ROCm Platform Installation Guide for Linux.

hipSPARSE can be installed on e.g. Ubuntu using

```
$ sudo apt-get update
$ sudo apt-get install hipsparse
```

Once installed, hipSPARSE can be used just like any other library with a C API. The header file will need to be included in the user code in order to make calls into hipSPARSE, and the hipSPARSE shared library will become link-time and run-time dependent for the user application.

1.2.3 Building hipSPARSE from source

Building from source is not necessary, as hipSPARSE can be used after installing the pre-built packages as described above. If desired, the following instructions can be used to build hipSPARSE from source. Furthermore, the following compile-time dependencies must be met

- rocSPARSE
- git
- CMake 3.5 or later
- AMD ROCm
- googletest (optional, for clients)

1.2.3.1 Download hipSPARSE

The hipSPARSE source code is available at the hipSPARSE GitHub page. Download the master branch using:

```
$ git clone -b master https://github.com/ROCmSoftwarePlatform/hipSPARSE.git
$ cd hipSPARSE
```

Below are steps to build different packages of the library, including dependencies and clients. It is recommended to install hipSPARSE using the *install.sh* script.

1.2.3.2 Using *install.sh* to build hipSPARSE with dependencies

The following table lists common uses of *install.sh* to build dependencies + library.

Command	Description
./install.sh -h	Print help information.
./install.sh -d	Build dependencies and library in your local directory. The *-d* flag only needs to be used once. For subsequent invocations of *install.sh* it is not necessary to rebuild the dependencies.
./install.sh	Build library in your local directory. It is assumed dependencies are available.
./install.sh -i	Build library, then build and install hipSPARSE package in */opt/rocm/hipsparse*. You will be prompted for sudo access. This will install for all users.

1.2.3.3 Using *install.sh* to build hipSPARSE with dependencies and clients

The client contains example code and unit tests. Common uses of *install.sh* to build them are listed in the table below.

Command	Description
./install.sh -h	Print help information.
./install.sh -dc	Build dependencies, library and client in your local directory. The *-d* flag only needs to be used once. For subsequent invocations of *install.sh* it is not necessary to rebuild the dependencies.
./install.sh -c	Build library and client in your local directory. It is assumed dependencies are available.
./install.sh -idc	Build library, dependencies and client, then build and install hipSPARSE package in */opt/rocm/hipsparse*. You will be prompted for sudo access. This will install for all users.
./install.sh -ic	Build library and client, then build and install hipSPARSE package in *opt/rocm/hipsparse*. You will be prompted for sudo access. This will install for all users.

1.2.3.4 Using individual commands to build hipSPARSE

CMake 3.5 or later is required in order to build hipSPARSE.

hipSPARSE can be built using the following commands:

```
# Create and change to build directory
$ mkdir -p build/release ; cd build/release

# Default install path is /opt/rocm, use -DCMAKE_INSTALL_PREFIX=<path> to adjust it
$ cmake ../..

# Compile hipSPARSE library
$ make -j$(nproc)

# Install hipSPARSE to /opt/rocm
$ make install
```

GoogleTest is required in order to build hipSPARSE clients.

hipSPARSE with dependencies and clients can be built using the following commands:

```
# Install googletest
$ mkdir -p build/release/deps ; cd build/release/deps
$ cmake ../../../deps
$ make -j$(nproc) install

# Change to build directory
$ cd ..

# Default install path is /opt/rocm, use -DCMAKE_INSTALL_PREFIX=<path> to adjust it
$ cmake ../.. -DBUILD_CLIENTS_TESTS=ON -DBUILD_CLIENTS_SAMPLES=ON

# Compile hipSPARSE library
$ make -j$(nproc)

# Install hipSPARSE to /opt/rocm
$ make install
```

1.2.3.5 Simple Test

You can test the installation by running one of the hipSPARSE examples, after successfully compiling the library with clients.

```
# Navigate to clients binary directory
$ cd hipSPARSE/build/release/clients/staging

# Execute hipSPARSE example
$ ./example_csrmv 1000
```

1.2.4 Supported Targets

Currently, hipSPARSE is supported under the following operating systems

- Ubuntu 18.04
- Ubuntu 20.04
- CentOS 7
- CentOS 8
- SLES 15

To compile and run hipSPARSE, AMD ROCm Platform is required.

1.3 Device and Stream Management

`hipSetDevice()` and `hipGetDevice()` are HIP device management APIs. They are NOT part of the hipSPARSE API.

1.3.1 Asynchronous Execution

All hipSPARSE library functions, unless otherwise stated, are non blocking and executed asynchronously with respect to the host. They may return before the actual computation has finished. To force synchronization, `hipDeviceSynchronize()` or `hipStreamSynchronize()` can be used. This will ensure that all previously executed hipSPARSE functions on the device / this particular stream have completed.

1.3.2 HIP Device Management

Before a HIP kernel invocation, users need to call `hipSetDevice()` to set a device, e.g. device 1. If users do not explicitly call it, the system by default sets it as device 0. Unless users explicitly call `hipSetDevice()` to set to another device, their HIP kernels are always launched on device 0.

The above is a HIP (and CUDA) device management approach and has nothing to do with hipSPARSE. hipSPARSE honors the approach above and assumes users have already set the device before a hipSPARSE routine call.

Once users set the device, they create a handle with *hipsparseCreate()*.

Subsequent hipSPARSE routines take this handle as an input parameter. hipSPARSE ONLY queries (by `hipGetDevice()`) the user's device; hipSPARSE does NOT set the device for users. If hipSPARSE does not see a valid device, it returns an error message. It is the users' responsibility to provide a valid device to hipSPARSE and ensure the device safety.

Users CANNOT switch devices between *hipsparseCreate()* and *hipsparseDestroy()*. If users want to change device, they must destroy the current handle and create another hipSPARSE handle.

1.3.3 HIP Stream Management

HIP kernels are always launched in a queue (also known as stream).

If users do not explicitly specify a stream, the system provides a default stream, maintained by the system. Users cannot create or destroy the default stream. However, users can freely create new streams (with `hipStreamCreate()`) and bind it to the hipSPARSE handle using *hipsparseGetVersion()*. HIP kernels are invoked in hipSPARSE routines. The hipSPARSE handle is always associated with a stream, and hipSPARSE passes its stream to the kernels inside the routine. One hipSPARSE routine only takes one stream in a single invocation. If users create a stream, they are responsible for destroying it.

1.3.4 Multiple Streams and Multiple Devices

If the system under test has multiple HIP devices, users can run multiple hipSPARSE handles concurrently, but can NOT run a single hipSPARSE handle on different discrete devices. Each handle is associated with a particular singular device, and a new handle should be created for each additional device.

1.4 Storage Formats

1.4.1 COO storage format

The Coordinate (COO) storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
coo_val	array of `nnz` elements containing the data (floating point).
coo_row_ind	array of `nnz` elements containing the row indices (integer).
coo_col_ind	array of `nnz` elements containing the column indices (integer).

The COO matrix is expected to be sorted by row indices and column indices per row. Furthermore, each pair of indices should appear only once. Consider the following 3×5 matrix and the corresponding COO structures, with $m = 3, n = 5$ and nnz $= 8$ using zero based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{coo\_val}[8] &= \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0\} \\ \text{coo\_row\_ind}[8] &= \{0, 0, 0, 1, 1, 2, 2, 2\} \\ \text{coo\_col\_ind}[8] &= \{0, 1, 3, 1, 2, 0, 3, 4\} \end{aligned}$$

1.4.2 COO (AoS) storage format

The Coordinate (COO) Array of Structure (AoS) storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
coo_val	array of `nnz` elements containing the data (floating point).
coo_ind	array of `2 * nnz` elements containing alternating row and column indices (integer).

The COO (AoS) matrix is expected to be sorted by row indices and column indices per row. Furthermore, each pair of indices should appear only once. Consider the following 3×5 matrix and the corresponding COO (AoS) structures, with $m = 3, n = 5$ and nnz $= 8$ using zero based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{coo\_val}[8] &= \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0\} \\ \text{coo\_ind}[16] &= \{0, 0, 0, 1, 0, 3, 1, 1, 1, 2, 2, 0, 2, 3, 2, 4\} \end{aligned}$$

1.4.3 CSR storage format

The Compressed Sparse Row (CSR) storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements (integer).
csr_val	array of `nnz` elements containing the data (floating point).
csr_row_ptr	array of `m+1` elements that point to the start of every row (integer).
csr_col_ind	array of `nnz` elements containing the column indices (integer).

The CSR matrix is expected to be sorted by column indices within each row. Furthermore, each pair of indices should appear only once. Consider the following 3×5 matrix and the corresponding CSR structures, with $m = 3, n = 5$ and nnz $= 8$ using one based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{csr\_val}[8] &= \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0\} \\ \text{csr\_row\_ptr}[4] &= \{1, 4, 6, 9\} \\ \text{csr\_col\_ind}[8] &= \{1, 2, 4, 2, 3, 1, 4, 5\} \end{aligned}$$

1.4.4 BSR storage format

The Block Compressed Sparse Row (BSR) storage format represents a $(mb \cdot \text{bsr\_dim}) \times (nb \cdot \text{bsr\_dim})$ matrix by

mb	number of block rows (integer)
nb	number of block columns (integer)
nnzb	number of non-zero blocks (integer)
bsr_val	array of `nnzb * bsr_dim * bsr_dim` elements containing the data (floating point). Blocks can be stored column-major or row-major.
bsr_-row_ptr	array of `mb+1` elements that point to the start of every block row (integer).
bsr_-col_ind	array of `nnzb` elements containing the block column indices (integer).
bsr_dim	dimension of each block (integer).

The BSR matrix is expected to be sorted by column indices within each row. If m or n are not evenly divisible by the block dimension, then zeros are padded to the matrix, such that $mb = (m + \text{bsr\_dim} - 1)/\text{bsr\_dim}$ and $nb = (n + \text{bsr\_dim} - 1)/\text{bsr\_dim}$. Consider the following 4×3 matrix and the corresponding BSR structures, with $\text{bsr\_dim} = 2, mb = 2, nb = 2$ and $\text{nnzb} = 4$ using zero based indexing and column-major storage:

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 \\ 3.0 & 0.0 & 4.0 \\ 5.0 & 6.0 & 0.0 \\ 7.0 & 0.0 & 8.0 \end{pmatrix}$$

with the blocks A_{ij}

$$A_{00} = \begin{pmatrix} 1.0 & 0.0 \\ 3.0 & 0.0 \end{pmatrix}, A_{01} = \begin{pmatrix} 2.0 & 0.0 \\ 4.0 & 0.0 \end{pmatrix}, A_{10} = \begin{pmatrix} 5.0 & 6.0 \\ 7.0 & 0.0 \end{pmatrix}, A_{11} = \begin{pmatrix} 0.0 & 0.0 \\ 8.0 & 0.0 \end{pmatrix}$$

such that

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

with arrays representation

$$\begin{aligned} \text{bsr\_val}[16] &= \{1.0, 3.0, 0.0, 0.0, 2.0, 4.0, 0.0, 0.0, 5.0, 7.0, 6.0, 0.0, 0.0, 8.0, 0.0, 0.0\} \\ \text{bsr\_row\_ptr}[3] &= \{0, 2, 4\} \\ \text{bsr\_col\_ind}[4] &= \{0, 1, 0, 1\} \end{aligned}$$

1.4.5 GEBSR storage format

The General Block Compressed Sparse Row (GEBSR) storage format represents a $(mb \cdot \text{bsr\_row\_dim}) \times (nb \cdot \text{bsr\_col\_dim})$ matrix by

mb	number of block rows (integer)
nb	number of block columns (integer)
nnzb	number of non-zero blocks (integer)
bsr_val	array of `nnzb * bsr_row_dim * bsr_col_dim` elements containing the data (floating point). Blocks can be stored column-major or row-major.
bsr_-row_ptr	array of `mb`+1 elements that point to the start of every block row (integer).
bsr_-col_ind	array of `nnzb` elements containing the block column indices (integer).
bsr_-row_dim	row dimension of each block (integer).
bsr_-col_dim	column dimension of each block (integer).

The GEBSR matrix is expected to be sorted by column indices within each row. If m is not evenly divisible by the row block dimension or n is not evenly divisible by the column block dimension, then zeros are padded to the matrix, such that $mb = (m + \text{bsr\_row\_dim} - 1)/\text{bsr\_row\_dim}$ and $nb = (n + \text{bsr\_col\_dim} - 1)/\text{bsr\_col\_dim}$. Consider the following 4×5 matrix and the corresponding GEBSR structures, with bsr_row_dim $= 2$, bsr_col_dim $= 3$, mb = 2, nb = 2` and nnzb $= 4$ using zero based indexing and column-major storage:

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

with the blocks A_{ij}

$$A_{00} = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 3.0 & 0.0 & 4.0 \end{pmatrix}, A_{01} = \begin{pmatrix} 2.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}, A_{10} = \begin{pmatrix} 5.0 & 6.0 & 0.0 \\ 0.0 & 0.0 & 8.0 \end{pmatrix}, A_{11} = \begin{pmatrix} 7.0 & 0.0 & 0.0 \\ 0.0 & 9.0 & 0.0 \end{pmatrix}$$

such that

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

with arrays representation

$$\begin{aligned} \text{bsr\_val}[24] &= \{1.0, 3.0, 0.0, 0.0, 0.0, 4.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 6.0, 0.0, 0.0, 8.0, 7.0, 0.0, 0.0, 9.0, 0.0, 0.0\} \\ \text{bsr\_row\_ptr}[3] &= \{0, 2, 4\} \\ \text{bsr\_col\_ind}[4] &= \{0, 1, 0, 1\} \end{aligned}$$

1.4.6 ELL storage format

The Ellpack-Itpack (ELL) storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
ell_width	maximum number of non-zero elements per row (integer)
ell_val	array of `m times ell_width` elements containing the data (floating point).
ell_col_ind	array of `m times ell_width` elements containing the column indices (integer).

The ELL matrix is assumed to be stored in column-major format. Rows with less than `ell_width` non-zero elements are padded with zeros (`ell_val`) and -1 (`ell_col_ind`). Consider the following 3×5 matrix and the corresponding

ELL structures, with $m = 3, n = 5$ and ell_width $= 3$ using zero based indexing:

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 4.0 & 5.0 & 0.0 & 0.0 \\ 6.0 & 0.0 & 0.0 & 7.0 & 8.0 \end{pmatrix}$$

where

$$\begin{aligned} \text{ell\_val}[9] &= \{1.0, 4.0, 6.0, 2.0, 5.0, 7.0, 3.0, 0.0, 8.0\} \\ \text{ell\_col\_ind}[9] &= \{0, 1, 0, 1, 2, 3, 3, -1, 4\} \end{aligned}$$

1.4.7 HYB storage format

The Hybrid (HYB) storage format represents a $m \times n$ matrix by

m	number of rows (integer).
n	number of columns (integer).
nnz	number of non-zero elements of the COO part (integer)
ell_width	maximum number of non-zero elements per row of the ELL part (integer)
ell_val	array of `m times ell_width` elements containing the ELL part data (floating point).
ell_col_ind	array of `m times ell_width` elements containing the ELL part column indices (integer).
coo_val	array of `nnz` elements containing the COO part data (floating point).
coo_row_ind	array of `nnz` elements containing the COO part row indices (integer).
coo_col_ind	array of `nnz` elements containing the COO part column indices (integer).

The HYB format is a combination of the ELL and COO sparse matrix formats. Typically, the regular part of the matrix is stored in ELL storage format, and the irregular part of the matrix is stored in COO storage format. Three different partitioning schemes can be applied when converting a CSR matrix to a matrix in HYB storage format. For further details on the partitioning schemes, see *hipsparseHybPartition_t*.

1.5 Types

1.5.1 hipsparseHandle_t

typedef void ***hipsparseHandle_t**

Handle to the hipSPARSE library context queue.

The hipSPARSE handle is a structure holding the hipSPARSE library context. It must be initialized using *hipsparseCreate()* and the returned handle must be passed to all subsequent library function calls. It should be destroyed at the end using *hipsparseDestroy()*.

1.5.2 hipsparseMatDescr_t

typedef void ***hipsparseMatDescr_t**

Descriptor of the matrix.

The hipSPARSE matrix descriptor is a structure holding all properties of a matrix. It must be initialized using *hipsparseCreateMatDescr()* and the returned descriptor must be passed to all subsequent library calls that involve the matrix. It should be destroyed at the end using *hipsparseDestroyMatDescr()*.

1.5.3 hipsparseHybMat_t

typedef void *hipsparseHybMat_t

HYB matrix storage format.

The hipSPARSE HYB matrix structure holds the HYB matrix. It must be initialized using *hipsparseCreateHybMat()* and the returned HYB matrix must be passed to all subsequent library calls that involve the matrix. It should be destroyed at the end using *hipsparseDestroyHybMat()*.

For more details on the HYB format, see *HYB storage format*.

1.5.4 hipsparseColorInfo_t

typedef void *hipsparseColorInfo_t

Coloring info.

The hipSPARSE ColorInfo structure holds the coloring information. It must be initialized using *hipsparseCreateColorInfo()* and the returned structure must be passed to all subsequent library calls that involve the coloring. It should be destroyed at the end using *hipsparseDestroyColorInfo()*.

1.5.5 bsrsv2Info_t

typedef struct bsrsv2Info *bsrsv2Info_t

1.5.6 bsrsm2Info_t

typedef struct bsrsm2Info *bsrsm2Info_t

1.5.7 bsrilu02Info_t

typedef struct bsrilu02Info *bsrilu02Info_t

1.5.8 bsric02Info_t

typedef struct bsric02Info *bsric02Info_t

1.5.9 csrsv2Info_t

typedef struct csrsv2Info *csrsv2Info_t

1.5.10 csrsm2Info_t

typedef struct csrsm2Info *csrsm2Info_t

1.5.11 csrilu02Info_t

typedef struct csrilu02Info *csrilu02Info_t

1.5.12 csric02Info_t

typedef struct csric02Info *csric02Info_t

1.5.13 csrgemm2Info_t

typedef struct csrgemm2Info *csrgemm2Info_t

1.5.14 pruneInfo_t

typedef struct pruneInfo *pruneInfo_t

1.5.15 csru2csrInfo_t

typedef struct csru2csrInfo *csru2csrInfo_t

1.5.16 hipsparseSpVecDescr_t

typedef void *hipsparseSpVecDescr_t

1.5.17 hipsparseSpMatDescr_t

typedef void *hipsparseSpMatDescr_t

1.5.18 hipsparseDnVecDescr_t

typedef void *hipsparseDnVecDescr_t

1.5.19 hipsparseDnMatDescr_t

typedef void *hipsparseDnMatDescr_t

1.5.20 hipsparseSpGEMMDescr_t

typedef struct hipsparseSpGEMMDescr *hipsparseSpGEMMDescr_t

1.5.21 hipsparseSpSVDescr_t

typedef struct hipsparseSpSVDescr *hipsparseSpSVDescr_t

1.5.22 hipsparseSpSMDescr_t

typedef struct hipsparseSpSMDescr *hipsparseSpSMDescr_t

1.5.23 hipsparseStatus_t

enum hipsparseStatus_t

List of hipsparse status codes definition.

This is a list of the *hipsparseStatus_t* types that are used by the hipSPARSE library.

Values:

enumerator HIPSPARSE_STATUS_SUCCESS

enumerator HIPSPARSE_STATUS_NOT_INITIALIZED

enumerator HIPSPARSE_STATUS_ALLOC_FAILED

enumerator HIPSPARSE_STATUS_INVALID_VALUE

enumerator **HIPSPARSE_STATUS_ARCH_MISMATCH**

enumerator **HIPSPARSE_STATUS_MAPPING_ERROR**

enumerator **HIPSPARSE_STATUS_EXECUTION_FAILED**

enumerator **HIPSPARSE_STATUS_INTERNAL_ERROR**

enumerator **HIPSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED**

enumerator **HIPSPARSE_STATUS_ZERO_PIVOT**

enumerator **HIPSPARSE_STATUS_NOT_SUPPORTED**

enumerator **HIPSPARSE_STATUS_INSUFFICIENT_RESOURCES**

1.5.24 hipsparsePointerMode_t

enum **hipsparsePointerMode_t**

Indicates if the pointer is device pointer or host pointer.

The *hipsparsePointerMode_t* indicates whether scalar values are passed by reference on the host or device. The *hipsparsePointerMode_t* can be changed by *hipsparseSetPointerMode()*. The currently used pointer mode can be obtained by *hipsparseGetPointerMode()*.

Values:

enumerator **HIPSPARSE_POINTER_MODE_HOST**

enumerator **HIPSPARSE_POINTER_MODE_DEVICE**

1.5.25 hipsparseAction_t

enum **hipsparseAction_t**

Specify where the operation is performed on.

The *hipsparseAction_t* indicates whether the operation is performed on the full matrix, or only on the sparsity pattern of the matrix.

Values:

enumerator **HIPSPARSE_ACTION_SYMBOLIC**

enumerator **HIPSPARSE_ACTION_NUMERIC**

1.5.26 hipsparseMatrixType_t

enum **hipsparseMatrixType_t**

Specify the matrix type.

The *hipsparseMatrixType_t* indices the type of a matrix. For a given *hipsparseMatDescr_t*, the *hipsparseMatrixType_t* can be set using *hipsparseSetMatType()*. The current *hipsparseMatrixType_t* of a matrix can be obtained by *hipsparseGetMatType()*.

Values:

enumerator **HIPSPARSE_MATRIX_TYPE_GENERAL**

enumerator **HIPSPARSE_MATRIX_TYPE_SYMMETRIC**

enumerator **HIPSPARSE_MATRIX_TYPE_HERMITIAN**

enumerator **HIPSPARSE_MATRIX_TYPE_TRIANGULAR**

1.5.27 hipsparseFillMode_t

enum **hipsparseFillMode_t**

Specify the matrix fill mode.

The *hipsparseFillMode_t* indicates whether the lower or the upper part is stored in a sparse triangular matrix. For a given *hipsparseMatDescr_t*, the *hipsparseFillMode_t* can be set using *hipsparseSetMatFillMode()*. The current *hipsparseFillMode_t* of a matrix can be obtained by *hipsparseGetMatFillMode()*.

Values:

enumerator **HIPSPARSE_FILL_MODE_LOWER**

enumerator **HIPSPARSE_FILL_MODE_UPPER**

1.5.28 hipsparseDiagType_t

enum **hipsparseDiagType_t**

Indicates if the diagonal entries are unity.

The *hipsparseDiagType_t* indicates whether the diagonal entries of a matrix are unity or not. If HIPSPARSE_DIAG_TYPE_UNIT is specified, all present diagonal values will be ignored. For a given *hipsparseMatDescr_t*, the *hipsparseDiagType_t* can be set using *hipsparseSetMatDiagType()*. The current *hipsparseDiagType_t* of a matrix can be obtained by *hipsparseGetMatDiagType()*.

Values:

enumerator **HIPSPARSE_DIAG_TYPE_NON_UNIT**

enumerator **HIPSPARSE_DIAG_TYPE_UNIT**

1.5.29 hipsparseIndexBase_t

enum **hipsparseIndexBase_t**

Specify the matrix index base.

The *hipsparseIndexBase_t* indicates the index base of the indices. For a given *hipsparseMatDescr_t*, the *hipsparseIndexBase_t* can be set using *hipsparseSetMatIndexBase()*. The current *hipsparseIndexBase_t* of a matrix can be obtained by *hipsparseGetMatIndexBase()*.

Values:

enumerator **HIPSPARSE_INDEX_BASE_ZERO**

enumerator **HIPSPARSE_INDEX_BASE_ONE**

1.5.30 hipsparseOperation_t

enum **hipsparseOperation_t**

Specify whether the matrix is to be transposed or not.

The *hipsparseOperation_t* indicates the operation performed with the given matrix.

Values:

enumerator **HIPSPARSE_OPERATION_NON_TRANSPOSE**

enumerator **HIPSPARSE_OPERATION_TRANSPOSE**

enumerator **HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE**

1.5.31 hipsparseHybPartition_t

enum **hipsparseHybPartition_t**

HYB matrix partitioning type.

The *hipsparseHybPartition_t* type indicates how the hybrid format partitioning between COO and ELL storage formats is performed.

Values:

enumerator **HIPSPARSE_HYB_PARTITION_AUTO**

enumerator **HIPSPARSE_HYB_PARTITION_USER**

enumerator **HIPSPARSE_HYB_PARTITION_MAX**

1.5.32 hipsparseSolvePolicy_t

enum **hipsparseSolvePolicy_t**

Specify policy in triangular solvers and factorizations.

The *hipsparseSolvePolicy_t* type indicates the solve policy for the triangular solve.

Values:

enumerator **HIPSPARSE_SOLVE_POLICY_NO_LEVEL**

enumerator **HIPSPARSE_SOLVE_POLICY_USE_LEVEL**

1.5.33 hipsparseSideMode_t

enum **hipsparseSideMode_t**

Values:

enumerator **HIPSPARSE_SIDE_LEFT**

enumerator **HIPSPARSE_SIDE_RIGHT**

1.5.34 hipsparseDirection_t

enum **hipsparseDirection_t**

Specify the matrix direction.

The *hipsparseDirection_t* indicates whether a dense matrix should be parsed by rows or by columns, assuming column-major storage.

Values:

enumerator **HIPSPARSE_DIRECTION_ROW**

enumerator **HIPSPARSE_DIRECTION_COLUMN**

1.5.35 hipsparseFormat_t

enum **hipsparseFormat_t**

Values:

enumerator **HIPSPARSE_FORMAT_CSR**

enumerator **HIPSPARSE_FORMAT_CSC**

enumerator **HIPSPARSE_FORMAT_COO**

enumerator **HIPSPARSE_FORMAT_COO_AOS**

enumerator **HIPSPARSE_FORMAT_BLOCKED_ELL**

1.5.36 hipsparseOrder_t

enum **hipsparseOrder_t**

Values:

enumerator **HIPSPARSE_ORDER_ROW**

enumerator **HIPSPARSE_ORDER_COLUMN**

enumerator **HIPSPARSE_ORDER_COL**

1.5.37 hipsparseIndextype_t

enum **hipsparseIndexType_t**

Values:

enumerator **HIPSPARSE_INDEX_16U**

enumerator **HIPSPARSE_INDEX_32I**

enumerator **HIPSPARSE_INDEX_64I**

1.5.38 hipsparseCsr2CscAlg_t

enum **hipsparseCsr2CscAlg_t**

Values:

enumerator **HIPSPARSE_CSR2CSC_ALG1**

enumerator **HIPSPARSE_CSR2CSC_ALG2**

1.5.39 hipsparseSpMVAlg_t

enum **hipsparseSpMVAlg_t**

Values:

enumerator **HIPSPARSE_MV_ALG_DEFAULT**

enumerator **HIPSPARSE_COOMV_ALG**

enumerator **HIPSPARSE_CSRMV_ALG1**

enumerator **HIPSPARSE_CSRMV_ALG2**

enumerator **HIPSPARSE_SPMV_ALG_DEFAULT**

enumerator **HIPSPARSE_SPMV_COO_ALG1**

enumerator **HIPSPARSE_SPMV_COO_ALG2**

enumerator **HIPSPARSE_SPMV_CSR_ALG1**

enumerator **HIPSPARSE_SPMV_CSR_ALG2**

1.5.40 hipsparseSpMMAlg_t

enum **hipsparseSpMMAlg_t**

Values:

enumerator **HIPSPARSE_MM_ALG_DEFAULT**

enumerator **HIPSPARSE_COOMM_ALG1**

enumerator **HIPSPARSE_COOMM_ALG2**

enumerator **HIPSPARSE_COOMM_ALG3**

enumerator **HIPSPARSE_CSRMM_ALG1**

enumerator **HIPSPARSE_SPMM_ALG_DEFAULT**

enumerator **HIPSPARSE_SPMM_COO_ALG1**

enumerator **HIPSPARSE_SPMM_COO_ALG2**

enumerator **HIPSPARSE_SPMM_COO_ALG3**

enumerator **HIPSPARSE_SPMM_COO_ALG4**

enumerator **HIPSPARSE_SPMM_CSR_ALG1**

enumerator **HIPSPARSE_SPMM_CSR_ALG2**

enumerator **HIPSPARSE_SPMM_BLOCKED_ELL_ALG1**

enumerator **HIPSPARSE_SPMM_CSR_ALG3**

1.5.41 hipsparseSparseToDenseAlg_t

enum **hipsparseSparseToDenseAlg_t**

Values:

enumerator **HIPSPARSE_SPARSETODENSE_ALG_DEFAULT**

1.5.42 hipsparseDenseToSparseAlg_t

enum **hipsparseDenseToSparseAlg_t**

Values:

enumerator **HIPSPARSE_DENSETOSPARSE_ALG_DEFAULT**

1.5.43 hipsparseSDDMMAlg_t

enum **hipsparseSDDMMAlg_t**

Values:

enumerator **HIPSPARSE_SDDMM_ALG_DEFAULT**

1.5.44 hipsparseSpSVAlg_t

enum **hipsparseSpSVAlg_t**

Values:

enumerator **HIPSPARSE_SPSV_ALG_DEFAULT**

1.5.45 hipsparseSpSMAlg_t

enum **hipsparseSpSMAlg_t**

Values:

enumerator **HIPSPARSE_SPSM_ALG_DEFAULT**

1.5.46 hipsparseSpMatAttribute_t

enum **hipsparseSpMatAttribute_t**

Values:

enumerator **HIPSPARSE_SPMAT_FILL_MODE**

enumerator **HIPSPARSE_SPMAT_DIAG_TYPE**

1.5.47 hipsparseSpGEMMAlg_t

enum **hipsparseSpGEMMAlg_t**

Values:

enumerator **HIPSPARSE_SPGEMM_DEFAULT**

enumerator **HIPSPARSE_SPGEMM_CSR_ALG_NONDETERMINISTIC**

enumerator **HIPSPARSE_SPGEMM_CSR_ALG_DETERMINISTIC**

1.6 Exported Sparse Functions

1.6.1 Auxiliary Functions

Function name
hipsparseCreate()
hipsparseDestroy()
hipsparseGetVersion()
hipsparseGetGitRevision()
hipsparseSetStream()
hipsparseGetStream()
hipsparseSetPointerMode()
hipsparseGetPointerMode()
hipsparseCreateMatDescr()
hipsparseDestroyMatDescr()
hipsparseCopyMatDescr()
hipsparseSetMatType()
hipsparseGetMatType()
hipsparseSetMatFillMode()
hipsparseGetMatFillMode()
hipsparseSetMatDiagType()
hipsparseGetMatDiagType()
hipsparseSetMatIndexBase()
hipsparseGetMatIndexBase()
hipsparseCreateHybMat()
hipsparseDestroyHybMat()
hipsparseCreateBsrsv2Info()
hipsparseDestroyBsrsv2Info()
hipsparseCreateBsrsm2Info()
hipsparseDestroyBsrsm2Info()
hipsparseCreateBsrilu02Info()
hipsparseDestroyBsrilu02Info()
hipsparseCreateBsric02Info()
hipsparseDestroyBsric02Info()
hipsparseCreateCsrsv2Info()
hipsparseDestroyCsrsv2Info()
hipsparseCreateCsrsm2Info()
hipsparseDestroyCsrsm2Info()
hipsparseCreateCsrilu02Info()
hipsparseDestroyCsrilu02Info()
hipsparseCreateCsric02Info()
hipsparseDestroyCsric02Info()
hipsparseCreateCsru2csrInfo()
hipsparseDestroyCsru2csrInfo()
hipsparseCreateColorInfo()
hipsparseDestroyColorInfo()
hipsparseCreateCsrgemm2Info()
hipsparseDestroyCsrgemm2Info()
hipsparseCreatePruneInfo()
hipsparseDestroyPruneInfo()

continues on next page

Table 1 – continued from previous page

hipsparseCreateSpVec()
hipsparseDestroySpVec()
hipsparseSpVecGet()
hipsparseSpVecGetIndexBase()
hipsparseSpVecGetValues()
hipsparseSpVecSetValues()
hipsparseCreateCoo()
hipsparseCreateCooAoS()
hipsparseCreateCsr()
hipsparseCreateCsc()
hipsparseCreateBlockedEll()
hipsparseDestroySpMat()
hipsparseCooGet()
hipsparseCooAoSGet()
hipsparseCsrGet()
hipsparseBlockedEllGet()
hipsparseCsrSetPointers()
hipsparseCscSetPointers()
hipsparseCooSetPointers()
hipsparseSpMatGetSize()
hipsparseSpMatGetFormat()
hipsparseSpMatGetIndexBase()
hipsparseSpMatGetValues()
hipsparseSpMatSetValues()
hipsparseSpMatGetAttribute()
hipsparseSpMatSetAttribute()
hipsparseCreateDnVec()
hipsparseDestroyDnVec()
hipsparseDnVecGet()
hipsparseDnVecGetValues()
hipsparseDnVecSetValues()
hipsparseCreateDnMat()
hipsparseDestroyDnMat()
hipsparseDnMatGet()
hipsparseDnMatGetValues()
hipsparseDnMatSetValues()

1.6.2 Sparse Level 1 Functions

Function name	single	double	single complex	double complex
hipsparseXaxpyi()	x	x	x	x
hipsparseXdoti()	x	x	x	x
hipsparseXdotci()			x	x
hipsparseXgthr()	x	x	x	x
hipsparseXgthrz()	x	x	x	x
hipsparseXroti()	x	x		
hipsparseXsctr()	x	x	x	x

1.6.3 Sparse Level 2 Functions

Function name	single	double	single complex	double complex
hipsparseXcsrmv()	x	x	x	x
hipsparseXcsrsv2_zeroPivot()				
hipsparseXcsrsv2_bufferSize()	x	x	x	x
hipsparseXcsrsv2_bufferSizeExt()	x	x	x	x
hipsparseXcsrsv2_analysis()	x	x	x	x
hipsparseXcsrsv2_solve()	x	x	x	x
hipsparseXhybmv()	x	x	x	x
hipsparseXbsrmv()	x	x	x	x
hipsparseXbsrxmv()	x	x	x	x
hipsparseXbsrsv2_zeroPivot()				
hipsparseXbsrsv2_bufferSize()	x	x	x	x
hipsparseXbsrsv2_bufferSizeExt()	x	x	x	x
hipsparseXbsrsv2_analysis()	x	x	x	x
hipsparseXbsrsv2_solve()	x	x	x	x
hipsparseXgemvi_bufferSize()	x	x	x	x
hipsparseXgemvi()	x	x	x	x

1.6.4 Sparse Level 3 Functions

Function name	single	double	single complex	double complex
hipsparseXbsrmm()	x	x	x	x
hipsparseXcsrmm()	x	x	x	x
hipsparseXcsrmm2()	x	x	x	x
hipsparseXbsrsm2_zeroPivot()				
hipsparseXbsrsm2_bufferSize()	x	x	x	x
hipsparseXbsrsm2_analysis()	x	x	x	x
hipsparseXbsrsm2_solve()	x	x	x	x
hipsparseXcsrsm2_zeroPivot()				
hipsparseXcsrsm2_bufferSizeExt()	x	x	x	x
hipsparseXcsrsm2_analysis()	x	x	x	x
hipsparseXcsrsm2_solve()	x	x	x	x
hipsparseXgemmi()	x	x	x	x

1.6.5 Sparse Extra Functions

Function name	single	double	single complex	double complex
hipsparseXcsrgeamNnz()				
hipsparseXcsrgeam()	x	x	x	x
hipsparseXcsrgeam2_bufferSizeExt()	x	x	x	x
hipsparseXcsrgeam2Nnz()				
hipsparseXcsrgeam2()	x	x	x	x
hipsparseXcsrgemmNnz()				
hipsparseXcsrgemm()	x	x	x	x
hipsparseXcsrgemm2_bufferSizeExt()	x	x	x	x
hipsparseXcsrgemm2Nnz()				
hipsparseXcsrgemm2()	x	x	x	x

1.6.6 Preconditioner Functions

Function name	single	double	single complex	double complex
hipsparseXbsrilu02_zeroPivot()				
hipsparseXbsrilu02_numericBoost()	x	x	x	x
hipsparseXbsrilu02_bufferSize()	x	x	x	x
hipsparseXbsrilu02_analysis()	x	x	x	x
hipsparseXbsrilu02()	x	x	x	x
hipsparseXcsrilu02_zeroPivot()				
hipsparseXcsrilu02_numericBoost()	x	x	x	x
hipsparseXcsrilu02_bufferSize()	x	x	x	x
hipsparseXcsrilu02_bufferSizeExt()	x	x	x	x
hipsparseXcsrilu02_analysis()	x	x	x	x
hipsparseXcsrilu02()	x	x	x	x
hipsparseXbsric02_zeroPivot()				
hipsparseXbsric02_bufferSize()	x	x	x	x
hipsparseXbsric02_analysis()	x	x	x	x
hipsparseXbsric02()	x	x	x	x
hipsparseXcsric02_zeroPivot()				
hipsparseXcsric02_bufferSize()	x	x	x	x
hipsparseXcsric02_bufferSizeExt()	x	x	x	x
hipsparseXcsric02_analysis()	x	x	x	x
hipsparseXcsric02()	x	x	x	x
hipsparseXgtsv2_bufferSizeExt()	x	x	x	x
hipsparseXgtsv2()	x	x	x	x
hipsparseXgtsv2_nopivot_bufferSizeExt()	x	x	x	x
hipsparseXgtsv2_nopivot()	x	x	x	x
hipsparseXgtsv2StridedBatch_bufferSizeExt()	x	x	x	x
hipsparseXgtsv2StridedBatch()	x	x	x	x
hipsparseXgtsvInterleavedBatch_bufferSizeExt()	x	x	x	x
hipsparseXgtsvInterleavedBatch()	x	x	x	x
hipsparseXgpsvInterleavedBatch_bufferSizeExt()	x	x	x	x
hipsparseXgpsvInterleavedBatch()	x	x	x	x

1.6.7 Conversion Functions

Function name	single	double	single complex	double complex
hipsparseXnnz()	x	x	x	x
hipsparseXdense2csr()	x	x	x	x
hipsparseXpruneDense2csr_bufferSize()	x	x		
hipsparseXpruneDense2csrNnz()	x	x		
hipsparseXpruneDense2csr()	x	x		
hipsparseXpruneDense2csrByPercentage_bufferSize()	x	x		
hipsparseXpruneDense2csrByPercentage_bufferSizeExt()	x	x		
hipsparseXpruneDense2csrNnzByPercentage()	x	x		
hipsparseXpruneDense2csrByPercentage()	x	x		
hipsparseXdense2csc()	x	x	x	x
hipsparseXcsr2dense()	x	x	x	x
hipsparseXcsc2dense()	x	x	x	x
hipsparseXcsr2bsrNnz()				
hipsparseXcsr2bsr()	x	x	x	x
hipsparseXnnz_compress()	x	x	x	x
hipsparseXcsr2coo()				
hipsparseXcsr2csc()	x	x	x	x
hipsparseXcsr2hyb()	x	x	x	x
hipsparseXgebsr2gebsc_bufferSize	x	x	x	x
hipsparseXgebsr2gebsc()	x	x	x	x
hipsparseXcsr2gebsr_bufferSize()	x	x	x	x
hipsparseXcsr2gebsrNnz()				
hipsparseXcsr2gebsr()	x	x	x	x
hipsparseXbsr2csr()	x	x	x	x
hipsparseXgebsr2csr()	x	x	x	x
hipsparseXcsr2csr_compress()	x	x	x	x
hipsparseXpruneCsr2csr_bufferSize()	x	x		
hipsparseXpruneCsr2csr_bufferSizeExt()	x	x		
hipsparseXpruneCsr2csrNnz()	x	x		
hipsparseXpruneCsr2csr()	x	x		
hipsparseXpruneCsr2csrByPercentage_bufferSize()	x	x		
hipsparseXpruneCsr2csrByPercentage_bufferSizeExt()	x	x		
hipsparseXpruneCsr2csrNnzByPercentage()	x	x		
hipsparseXpruneCsr2csrByPercentage()	x	x		
hipsparseXhyb2csr()	x	x	x	x
hipsparseXcoo2csr()				
hipsparseCreateIdentityPermutation()				
hipsparseXcsrsort_bufferSizeExt()				
hipsparseXcsrsort()				
hipsparseXcscsort_bufferSizeExt()				
hipsparseXcscsort()				
hipsparseXcoosort_bufferSizeExt()				
hipsparseXcoosortByRow()				
hipsparseXcoosortByColumn()				
hipsparseXgebsr2gebsr_bufferSize()	x	x	x	x
hipsparseXgebsr2gebsrNnz()				
hipsparseXgebsr2gebsr()	x	x	x	x

continues on next page

Table 3 – continued from previous page

Function name	single	double	single complex	double complex
hipsparseXcsru2csr_bufferSizeExt()	x	x	x	x
hipsparseXcsru2csr()	x	x	x	x
hipsparseXcsr2csru()	x	x	x	x

1.6.8 Reordering Functions

Function name	single	double	single complex	double complex
hipsparseXcsrcolor()	x	x	x	x

1.6.9 Sparse Generic Functions

Function name	single	double	single complex	double complex
hipsparseAxpby()	x	x	x	x
hipsparseGather()	x	x	x	x
hipsparseScatter()	x	x	x	x
hipsparseRot()	x	x	x	x
hipsparseSparseToDense_bufferSize()	x	x	x	x
hipsparseSparseToDense()	x	x	x	x
hipsparseDenseToSparse_bufferSize()	x	x	x	x
hipsparseDenseToSparse_analysis()	x	x	x	x
hipsparseDenseToSparse_convert()	x	x	x	x
hipsparseSpVV_bufferSize()	x	x	x	x
hipsparseSpVV()	x	x	x	x
hipsparseSpMV_bufferSize()	x	x	x	x
hipsparseSpMV()	x	x	x	x
hipsparseSpMM_bufferSize()	x	x	x	x
hipsparseSpMM_preprocess()	x	x	x	x
hipsparseSpMM()	x	x	x	x
hipsparseSpGEMM_createDescr()	x	x	x	x
hipsparseSpGEMM_destroyDescr()	x	x	x	x
hipsparseSpGEMM_workEstimation()	x	x	x	x
hipsparseSpGEMM_compute()	x	x	x	x
hipsparseSpGEMM_copy()	x	x	x	x
hipsparseSDDMM_bufferSize()	x	x	x	x
hipsparseSDDMM_preprocess()	x	x	x	x
hipsparseSDDMM()	x	x	x	x
hipsparseSpSV_createDescr()	x	x	x	x
hipsparseSpSV_destroyDescr()	x	x	x	x
hipsparseSpSV_bufferSize()	x	x	x	x
hipsparseSpSV_analysis()	x	x	x	x
hipsparseSpSV_solve()	x	x	x	x
hipsparseSpSM_createDescr()	x	x	x	x
hipsparseSpSM_destroyDescr()	x	x	x	x
hipsparseSpSM_bufferSize()	x	x	x	x
hipsparseSpSM_analysis()	x	x	x	x
hipsparseSpSM_solve()	x	x	x	x

1.6.10 Storage schemes and indexing base

hipSPARSE supports 0 and 1 based indexing. The index base is selected by the *hipsparseIndexBase_t* type which is either passed as standalone parameter or as part of the *hipsparseMatDescr_t* type.

Furthermore, dense vectors are represented with a 1D array, stored linearly in memory. Sparse vectors are represented by a 1D data array stored linearly in memory that hold all non-zero elements and a 1D indexing array stored linearly in memory that hold the positions of the corresponding non-zero elements.

1.6.11 Pointer mode

The auxiliary functions *hipsparseSetPointerMode()* and *hipsparseGetPointerMode()* are used to set and get the value of the state variable *hipsparsePointerMode_t*. If *hipsparsePointerMode_t* is equal to HIPSPARSE_POINTER_MODE_HOST, then scalar parameters must be allocated on the host. If *hipsparsePointerMode_t* is equal to HIPSPARSE_POINTER_MODE_DEVICE, then scalar parameters must be allocated on the device.

There are two types of scalar parameter:

1. Scaling parameters, such as *alpha* and *beta* used in e.g. *hipsparseScsrmv()*, *hipsparseSbsrmv()*, ...
2. Scalar results from functions such as *hipsparseSdoti()*, *hipsparseCdotci()*, ...

For scalar parameters such as alpha and beta, memory can be allocated on the host heap or stack, when *hipsparsePointerMode_t* is equal to HIPSPARSE_POINTER_MODE_HOST. The kernel launch is asynchronous, and if the scalar parameter is on the heap, it can be freed after the return from the kernel launch. When *hipsparsePointerMode_t* is equal to HIPSPARSE_POINTER_MODE_DEVICE, the scalar parameter must not be changed till the kernel completes.

For scalar results, when *hipsparsePointerMode_t* is equal to HIPSPARSE_POINTER_MODE_HOST, the function blocks the CPU till the GPU has copied the result back to the host. Using *hipsparsePointerMode_t* equal to HIPSPARSE_POINTER_MODE_DEVICE, the function will return after the asynchronous launch. Similarly to vector and matrix results, the scalar result is only available when the kernel has completed execution.

1.6.12 Asynchronous API

Except a functions having memory allocation inside preventing asynchronicity, all hipSPARSE functions are configured to operate in non-blocking fashion with respect to CPU, meaning these library functions return immediately.

1.7 Sparse Auxiliary Functions

This module holds all sparse auxiliary functions.

The functions that are contained in the auxiliary module describe all available helper functions that are required for subsequent library calls.

1.7.1 hipsparseCreate()

hipsparseStatus_t **hipsparseCreate**(*hipsparseHandle_t* *handle)

Create a hipsparse handle.

`hipsparseCreate` creates the hipSPARSE library context. It must be initialized before any other hipSPARSE API function is invoked and must be passed to all subsequent library function calls. The handle should be destroyed at the end using *hipsparseDestroy()*.

1.7.2 hipsparseDestroy()

hipsparseStatus_t **hipsparseDestroy**(*hipsparseHandle_t* handle)

Destroy a hipsparse handle.

`hipsparseDestroy` destroys the hipSPARSE library context and releases all resources used by the hipSPARSE library.

1.7.3 hipsparseGetVersion()

hipsparseStatus_t **hipsparseGetVersion**(*hipsparseHandle_t* handle, int *version)

Get hipSPARSE version.

`hipsparseGetVersion` gets the hipSPARSE library version number.

- patch = version % 100
- minor = version / 100 % 1000
- major = version / 100000

1.7.4 hipsparseGetGitRevision()

hipsparseStatus_t **hipsparseGetGitRevision**(*hipsparseHandle_t* handle, char *rev)

Get hipSPARSE git revision.

`hipsparseGetGitRevision` gets the hipSPARSE library git commit revision (SHA-1).

1.7.5 hipsparseSetStream()

hipsparseStatus_t **hipsparseSetStream**(*hipsparseHandle_t* handle, hipStream_t streamId)

Specify user defined HIP stream.

`hipsparseSetStream` specifies the stream to be used by the hipSPARSE library context and all subsequent function calls.

1.7.6 hipsparseGetStream()

hipsparseStatus_t **hipsparseGetStream**(*hipsparseHandle_t* handle, hipStream_t *streamId)

Get current stream from library context.

`hipsparseGetStream` gets the hipSPARSE library context stream which is currently used for all subsequent function calls.

1.7.7 hipsparseSetPointerMode()

hipsparseStatus_t **hipsparseSetPointerMode**(*hipsparseHandle_t* handle, *hipsparsePointerMode_t* mode)

Specify pointer mode.

`hipsparseSetPointerMode` specifies the pointer mode to be used by the hipSPARSE library context and all subsequent function calls. By default, all values are passed by reference on the host. Valid pointer modes are HIPSPARSE_POINTER_MODE_HOST or `HIPSPARSE_POINTER_MODE_DEVICE`.

1.7.8 hipsparseGetPointerMode()

hipsparseStatus_t **hipsparseGetPointerMode**(*hipsparseHandle_t* handle, *hipsparsePointerMode_t* *mode)

Get current pointer mode from library context.

`hipsparseGetPointerMode` gets the hipSPARSE library context pointer mode which is currently used for all subsequent function calls.

1.7.9 hipsparseCreateMatDescr()

hipsparseStatus_t **hipsparseCreateMatDescr**(*hipsparseMatDescr_t* *descrA)

Create a matrix descriptor.

`hipsparseCreateMatDescr` creates a matrix descriptor. It initializes *hipsparseMatrixType_t* to HIPSPARSE_MATRIX_TYPE_GENERAL and *hipsparseIndexBase_t* to HIPSPARSE_INDEX_BASE_ZERO. It should be destroyed at the end using *hipsparseDestroyMatDescr()*.

1.7.10 hipsparseDestroyMatDescr()

hipsparseStatus_t **hipsparseDestroyMatDescr**(*hipsparseMatDescr_t* descrA)

Destroy a matrix descriptor.

`hipsparseDestroyMatDescr` destroys a matrix descriptor and releases all resources used by the descriptor.

1.7.11 hipsparseCopyMatDescr()

hipsparseStatus_t **hipsparseCopyMatDescr**(*hipsparseMatDescr_t* dest, const *hipsparseMatDescr_t* src)

Copy a matrix descriptor.

`hipsparseCopyMatDescr` copies a matrix descriptor. Both, source and destination matrix descriptors must be initialized prior to calling `hipsparseCopyMatDescr`.

1.7.12 hipsparseSetMatType()

hipsparseStatus_t **hipsparseSetMatType**(*hipsparseMatDescr_t* descrA, *hipsparseMatrixType_t* type)

Specify the matrix type of a matrix descriptor.

`hipsparseSetMatType` sets the matrix type of a matrix descriptor. Valid matrix types are HIPSPARSE_MATRIX_TYPE_GENERAL, HIPSPARSE_MATRIX_TYPE_SYMMETRIC, HIPSPARSE_MATRIX_TYPE_HERMITIAN or HIPSPARSE_MATRIX_TYPE_TRIANGULAR.

1.7.13 hipsparseGetMatType()

hipsparseMatrixType_t **hipsparseGetMatType**(const *hipsparseMatDescr_t* descrA)

Get the matrix type of a matrix descriptor.

`hipsparseGetMatType` returns the matrix type of a matrix descriptor.

1.7.14 hipsparseSetMatFillMode()

hipsparseStatus_t **hipsparseSetMatFillMode**(*hipsparseMatDescr_t* descrA, *hipsparseFillMode_t* fillMode)

Specify the matrix fill mode of a matrix descriptor.

`hipsparseSetMatFillMode` sets the matrix fill mode of a matrix descriptor. Valid fill modes are HIPSPARSE_FILL_MODE_LOWER or HIPSPARSE_FILL_MODE_UPPER.

1.7.15 hipsparseGetMatFillMode()

hipsparseFillMode_t **hipsparseGetMatFillMode**(const *hipsparseMatDescr_t* descrA)

Get the matrix fill mode of a matrix descriptor.

`hipsparseGetMatFillMode` returns the matrix fill mode of a matrix descriptor.

1.7.16 hipsparseSetMatDiagType()

hipsparseStatus_t **hipsparseSetMatDiagType**(*hipsparseMatDescr_t* descrA, *hipsparseDiagType_t* diagType)

Specify the matrix diagonal type of a matrix descriptor.

`hipsparseSetMatDiagType` sets the matrix diagonal type of a matrix descriptor. Valid diagonal types are HIPSPARSE_DIAG_TYPE_UNIT or HIPSPARSE_DIAG_TYPE_NON_UNIT.

1.7.17 hipsparseGetMatDiagType()

hipsparseDiagType_t **hipsparseGetMatDiagType**(const *hipsparseMatDescr_t* descrA)

Get the matrix diagonal type of a matrix descriptor.

`hipsparseGetMatDiagType` returns the matrix diagonal type of a matrix descriptor.

1.7.18 hipsparseSetMatIndexBase()

hipsparseStatus_t **hipsparseSetMatIndexBase**(*hipsparseMatDescr_t* descrA, *hipsparseIndexBase_t* base)

Specify the index base of a matrix descriptor.

`hipsparseSetMatIndexBase` sets the index base of a matrix descriptor. Valid options are HIPSPARSE_INDEX_BASE_ZERO or HIPSPARSE_INDEX_BASE_ONE.

1.7.19 hipsparseGetMatIndexBase()

hipsparseIndexBase_t **hipsparseGetMatIndexBase**(const *hipsparseMatDescr_t* descrA)

Get the index base of a matrix descriptor.

`hipsparseGetMatIndexBase` returns the index base of a matrix descriptor.

1.7.20 hipsparseCreateHybMat()

hipsparseStatus_t **hipsparseCreateHybMat**(*hipsparseHybMat_t* *hybA)

Create a `HYB` matrix structure.

`hipsparseCreateHybMat` creates a structure that holds the matrix in `HYB` storage format. It should be destroyed at the end using *hipsparseDestroyHybMat()*.

1.7.21 hipsparseDestroyHybMat()

hipsparseStatus_t **hipsparseDestroyHybMat**(*hipsparseHybMat_t* hybA)

Destroy a `HYB` matrix structure.

`hipsparseDestroyHybMat` destroys a `HYB` structure.

1.7.22 hipsparseCreateBsrsv2Info()

hipsparseStatus_t **hipsparseCreateBsrsv2Info**(*bsrsv2Info_t* *info)

Create a bsrsv2 info structure.

`hipsparseCreateBsrsv2Info` creates a structure that holds the bsrsv2 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyBsrsv2Info()*.

1.7.23 hipsparseDestroyBsrsv2Info()

hipsparseStatus_t **hipsparseDestroyBsrsv2Info**(*bsrsv2Info_t* info)

Destroy a bsrsv2 info structure.

`hipsparseDestroyBsrsv2Info` destroys a bsrsv2 info structure.

1.7.24 hipsparseCreateBsrsm2Info()

hipsparseStatus_t **hipsparseCreateBsrsm2Info**(*bsrsm2Info_t* *info)

Create a bsrsm2 info structure.

`hipsparseCreateBsrsm2Info` creates a structure that holds the bsrsm2 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyBsrsm2Info()*.

1.7.25 hipsparseDestroyBsrsm2Info()

hipsparseStatus_t **hipsparseDestroyBsrsm2Info**(*bsrsm2Info_t* info)

Destroy a bsrsm2 info structure.

`hipsparseDestroyBsrsm2Info` destroys a bsrsm2 info structure.

1.7.26 hipsparseCreateBsrilu02Info()

hipsparseStatus_t **hipsparseCreateBsrilu02Info**(*bsrilu02Info_t* *info)

Create a bsrilu02 info structure.

`hipsparseCreateBsrilu02Info` creates a structure that holds the bsrilu02 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyBsrilu02Info()*.

1.7.27 hipsparseDestroyBsrilu02Info()

hipsparseStatus_t **hipsparseDestroyBsrilu02Info**(*bsrilu02Info_t* info)

Destroy a bsrilu02 info structure.

`hipsparseDestroyBsrilu02Info` destroys a bsrilu02 info structure.

1.7.28 hipsparseCreateBsric02Info()

hipsparseStatus_t **hipsparseCreateBsric02Info**(*bsric02Info_t* *info)

Create a bsric02 info structure.

`hipsparseCreateBsric02Info` creates a structure that holds the bsric02 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyBsric02Info()*.

1.7.29 hipsparseDestroyBsric02Info()

hipsparseStatus_t **hipsparseDestroyBsric02Info**(*bsric02Info_t* info)

Destroy a bsric02 info structure.

`hipsparseDestroyBsric02Info` destroys a bsric02 info structure.

1.7.30 hipsparseCreateCsrsv2Info()

hipsparseStatus_t **hipsparseCreateCsrsv2Info**(*csrsv2Info_t* *info)

Create a csrsv2 info structure.

`hipsparseCreateCsrsv2Info` creates a structure that holds the csrsv2 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsrsv2Info()*.

1.7.31 hipsparseDestroyCsrsv2Info()

hipsparseStatus_t **hipsparseDestroyCsrsv2Info**(*csrsv2Info_t* info)

Destroy a csrsv2 info structure.

`hipsparseDestroyCsrsv2Info` destroys a csrsv2 info structure.

1.7.32 hipsparseCreateCsrsm2Info()

hipsparseStatus_t **hipsparseCreateCsrsm2Info**(*csrsm2Info_t* *info)

Create a csrsm2 info structure.

`hipsparseCreateCsrsm2Info` creates a structure that holds the csrsm2 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsrsm2Info()*.

1.7.33 hipsparseDestroyCsrsm2Info()

hipsparseStatus_t **hipsparseDestroyCsrsm2Info**(*csrsm2Info_t* info)

Destroy a csrsm2 info structure.

`hipsparseDestroyCsrsm2Info` destroys a csrsm2 info structure.

1.7.34 hipsparseCreateCsrilu02Info()

hipsparseStatus_t **hipsparseCreateCsrilu02Info**(*csrilu02Info_t* *info)

Create a csrilu02 info structure.

`hipsparseCreateCsrilu02Info` creates a structure that holds the csrilu02 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsrilu02Info()*.

1.7.35 hipsparseDestroyCsrilu02Info()

hipsparseStatus_t **hipsparseDestroyCsrilu02Info**(*csrilu02Info_t* info)

Destroy a csrilu02 info structure.

`hipsparseDestroyCsrilu02Info` destroys a csrilu02 info structure.

1.7.36 hipsparseCreateCsric02Info()

hipsparseStatus_t **hipsparseCreateCsric02Info**(*csric02Info_t* *info)

Create a csric02 info structure.

`hipsparseCreateCsric02Info` creates a structure that holds the csric02 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsric02Info()*.

1.7.37 hipsparseDestroyCsric02Info()

hipsparseStatus_t **hipsparseDestroyCsric02Info**(*csric02Info_t* info)

Destroy a csric02 info structure.

`hipsparseDestroyCsric02Info` destroys a csric02 info structure.

1.7.38 hipsparseCreateCsru2csrInfo()

hipsparseStatus_t **hipsparseCreateCsru2csrInfo**(*csru2csrInfo_t* *info)

Create a csru2csr info structure.

`hipsparseCreateCsru2csrInfo` creates a structure that holds the csru2csr info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsru2csrInfo()*.

1.7.39 hipsparseDestroyCsru2csrInfo()

hipsparseStatus_t **hipsparseDestroyCsru2csrInfo**(*csru2csrInfo_t* info)

Destroy a csru2csr info structure.

`hipsparseDestroyCsru2csrInfo` destroys a csru2csr info structure.

1.7.40 hipsparseCreateColorInfo()

hipsparseStatus_t **hipsparseCreateColorInfo**(*hipsparseColorInfo_t* *info)

Create a color info structure.

`hipsparseCreateColorInfo` creates a structure that holds the color info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyColorInfo()*.

1.7.41 hipsparseDestroyColorInfo()

hipsparseStatus_t **hipsparseDestroyColorInfo**(*hipsparseColorInfo_t* info)

Destroy a color info structure.

`hipsparseDestroyColorInfo` destroys a color info structure.

1.7.42 hipsparseCreateCsrgemm2Info()

hipsparseStatus_t **hipsparseCreateCsrgemm2Info**(*csrgemm2Info_t* *info)

Create a csrgemm2 info structure.

`hipsparseCreateCsrgemm2Info` creates a structure that holds the csrgemm2 info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyCsrgemm2Info()*.

1.7.43 hipsparseDestroyCsrgemm2Info()

hipsparseStatus_t **hipsparseDestroyCsrgemm2Info**(*csrgemm2Info_t* info)

Destroy a csrgemm2 info structure.

`hipsparseDestroyCsrgemm2Info` destroys a csrgemm2 info structure.

1.7.44 hipsparseCreatePruneInfo()

hipsparseStatus_t **hipsparseCreatePruneInfo**(*pruneInfo_t* *info)

Create a prune info structure.

`hipsparseCreatePruneInfo` creates a structure that holds the prune info data that is gathered during the analysis routines available. It should be destroyed at the end using *hipsparseDestroyPruneInfo()*.

1.7.45 hipsparseDestroyPruneInfo()

hipsparseStatus_t **hipsparseDestroyPruneInfo**(*pruneInfo_t* info)

Destroy a prune info structure.

`hipsparseDestroyPruneInfo` destroys a prune info structure.

1.7.46 hipsparseCreateSpVec()

hipsparseStatus_t **hipsparseCreateSpVec**(*hipsparseSpVecDescr_t* *spVecDescr, int64_t size, int64_t nnz, void *indices, void *values, *hipsparseIndexType_t* idxType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.47 hipsparseDestroySpVec()

hipsparseStatus_t **hipsparseDestroySpVec**(*hipsparseSpVecDescr_t* spVecDescr)

1.7.48 hipsparseSpVecGet()

hipsparseStatus_t **hipsparseSpVecGet**(const *hipsparseSpVecDescr_t* spVecDescr, int64_t *size, int64_t *nnz, void **indices, void **values, *hipsparseIndexType_t* *idxType, *hipsparseIndexBase_t* *idxBase, hipDataType *valueType)

1.7.49 hipsparseSpVecGetIndexBase()

hipsparseStatus_t **hipsparseSpVecGetIndexBase**(const *hipsparseSpVecDescr_t* spVecDescr, *hipsparseIndexBase_t* *idxBase)

1.7.50 hipsparseSpVecGetValues()

hipsparseStatus_t **hipsparseSpVecGetValues**(const *hipsparseSpVecDescr_t* spVecDescr, void **values)

1.7.51 hipsparseSpVecSetValues()

hipsparseStatus_t **hipsparseSpVecSetValues**(*hipsparseSpVecDescr_t* spVecDescr, void *values)

1.7.52 hipsparseCreateCoo()

hipsparseStatus_t **hipsparseCreateCoo**(*hipsparseSpMatDescr_t* *spMatDescr, int64_t rows, int64_t cols, int64_t nnz, void *cooRowInd, void *cooColInd, void *cooValues, *hipsparseIndexType_t* cooIdxType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.53 hipsparseCreateCooAoS()

hipsparseStatus_t **hipsparseCreateCooAoS**(*hipsparseSpMatDescr_t* *spMatDescr, int64_t rows, int64_t cols, int64_t nnz, void *cooInd, void *cooValues, *hipsparseIndexType_t* cooIdxType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.54 hipsparseCreateCsr()

hipsparseStatus_t **hipsparseCreateCsr**(*hipsparseSpMatDescr_t* *spMatDescr, int64_t rows, int64_t cols, int64_t nnz, void *csrRowOffsets, void *csrColInd, void *csrValues, *hipsparseIndexType_t* csrRowOffsetsType, *hipsparseIndexType_t* csrColIndType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.55 hipsparseCreateCsc()

hipsparseStatus_t **hipsparseCreateCsc**(*hipsparseSpMatDescr_t* *spMatDescr, int64_t rows, int64_t cols, int64_t nnz, void *cscColOffsets, void *cscRowInd, void *cscValues, *hipsparseIndexType_t* cscColOffsetsType, *hipsparseIndexType_t* cscRowIndType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.56 hipsparseCreateBlockedEll()

hipsparseStatus_t **hipsparseCreateBlockedEll**(*hipsparseSpMatDescr_t* *spMatDescr, int64_t rows, int64_t cols, int64_t ellBlockSize, int64_t ellCols, void *ellColInd, void *ellValue, *hipsparseIndexType_t* ellIdxType, *hipsparseIndexBase_t* idxBase, hipDataType valueType)

1.7.57 hipsparseDestroySpMat()

hipsparseStatus_t **hipsparseDestroySpMat**(*hipsparseSpMatDescr_t* spMatDescr)

1.7.58 hipsparseCooGet()

hipsparseStatus_t **hipsparseCooGet**(const *hipsparseSpMatDescr_t* spMatDescr, int64_t *rows, int64_t *cols, int64_t *nnz, void **cooRowInd, void **cooColInd, void **cooValues, *hipsparseIndexType_t* *idxType, *hipsparseIndexBase_t* *idxBase, hipDataType *valueType)

1.7.59 hipsparseCooAoSGet()

hipsparseStatus_t **hipsparseCooAoSGet**(const *hipsparseSpMatDescr_t* spMatDescr, int64_t *rows, int64_t *cols, int64_t *nnz, void **cooInd, void **cooValues, *hipsparseIndexType_t* *idxType, *hipsparseIndexBase_t* *idxBase, hipDataType *valueType)

1.7.60 hipsparseCsrGet()

hipsparseStatus_t **hipsparseCsrGet**(const *hipsparseSpMatDescr_t* spMatDescr, int64_t *rows, int64_t *cols, int64_t *nnz, void **csrRowOffsets, void **csrColInd, void **csrValues, *hipsparseIndexType_t* *csrRowOffsetsType, *hipsparseIndexType_t* *csrColIndType, *hipsparseIndexBase_t* *idxBase, hipDataType *valueType)

1.7.61 hipsparseBlockedEllGet()

hipsparseStatus_t **hipsparseBlockedEllGet**(const *hipsparseSpMatDescr_t* spMatDescr, int64_t *rows, int64_t *cols, int64_t *ellBlockSize, int64_t *ellCols, void **ellColInd, void **ellValue, *hipsparseIndexType_t* *ellIdxType, *hipsparseIndexBase_t* *idxBase, hipDataType *valueType)

1.7.62 hipsparseCsrSetPointers()

hipsparseStatus_t **hipsparseCsrSetPointers**(*hipsparseSpMatDescr_t* spMatDescr, void *csrRowOffsets, void *csrColInd, void *csrValues)

1.7.63 hipsparseCscSetPointers()

hipsparseStatus_t **hipsparseCscSetPointers**(*hipsparseSpMatDescr_t* spMatDescr, void *cscColOffsets, void *cscRowInd, void *cscValues)

1.7.64 hipsparseCooSetPointers()

hipsparseStatus_t **hipsparseCooSetPointers**(*hipsparseSpMatDescr_t* spMatDescr, void *cooRowInd, void *cooColInd, void *cooValues)

1.7.65 hipsparseSpMatGetSize()

hipsparseStatus_t **hipsparseSpMatGetSize**(*hipsparseSpMatDescr_t* spMatDescr, int64_t *rows, int64_t *cols, int64_t *nnz)

1.7.66 hipsparseSpMatGetFormat()

hipsparseStatus_t **hipsparseSpMatGetFormat**(const *hipsparseSpMatDescr_t* spMatDescr, *hipsparseFormat_t* *format)

1.7.67 hipsparseSpMatGetIndexBase()

hipsparseStatus_t **hipsparseSpMatGetIndexBase**(const *hipsparseSpMatDescr_t* spMatDescr, *hipsparseIndexBase_t* *idxBase)

1.7.68 hipsparseSpMatGetValues()

hipsparseStatus_t **hipsparseSpMatGetValues**(*hipsparseSpMatDescr_t* spMatDescr, void **values)

1.7.69 hipsparseSpMatSetValues()

hipsparseStatus_t **hipsparseSpMatSetValues**(*hipsparseSpMatDescr_t* spMatDescr, void *values)

1.7.70 hipsparseSpMatGetAttribute()

hipsparseStatus_t **hipsparseSpMatGetAttribute**(*hipsparseSpMatDescr_t* spMatDescr,
hipsparseSpMatAttribute_t attribute, void *data, size_t
dataSize)

1.7.71 hipsparseSpMatSetAttribute()

hipsparseStatus_t **hipsparseSpMatSetAttribute**(*hipsparseSpMatDescr_t* spMatDescr,
hipsparseSpMatAttribute_t attribute, const void *data, size_t
dataSize)

1.7.72 hipsparseCreateDnVec()

hipsparseStatus_t **hipsparseCreateDnVec**(*hipsparseDnVecDescr_t* *dnVecDescr, int64_t size, void *values,
hipDataType valueType)

1.7.73 hipsparseDestroyDnVec()

hipsparseStatus_t **hipsparseDestroyDnVec**(*hipsparseDnVecDescr_t* dnVecDescr)

1.7.74 hipsparseDnVecGet()

hipsparseStatus_t **hipsparseDnVecGet**(const *hipsparseDnVecDescr_t* dnVecDescr, int64_t *size, void **values,
hipDataType *valueType)

1.7.75 hipsparseDnVecGetValues()

hipsparseStatus_t **hipsparseDnVecGetValues**(const *hipsparseDnVecDescr_t* dnVecDescr, void **values)

1.7.76 hipsparseDnVecSetValues()

hipsparseStatus_t **hipsparseDnVecSetValues**(*hipsparseDnVecDescr_t* dnVecDescr, void *values)

1.7.77 hipsparseCreateDnMat()

hipsparseStatus_t **hipsparseCreateDnMat**(*hipsparseDnMatDescr_t* *dnMatDescr, int64_t rows, int64_t cols, int64_t ld, void *values, hipDataType valueType, *hipsparseOrder_t* order)

1.7.78 hipsparseDestroyDnMat()

hipsparseStatus_t **hipsparseDestroyDnMat**(*hipsparseDnMatDescr_t* dnMatDescr)

1.7.79 hipsparseDnMatGet()

hipsparseStatus_t **hipsparseDnMatGet**(const *hipsparseDnMatDescr_t* dnMatDescr, int64_t *rows, int64_t *cols, int64_t *ld, void **values, hipDataType *valueType, *hipsparseOrder_t* *order)

1.7.80 hipsparseDnMatGetValues()

hipsparseStatus_t **hipsparseDnMatGetValues**(const *hipsparseDnMatDescr_t* dnMatDescr, void **values)

1.7.81 hipsparseDnMatSetValues()

hipsparseStatus_t **hipsparseDnMatSetValues**(*hipsparseDnMatDescr_t* dnMatDescr, void *values)

1.8 Sparse Level 1 Functions

The sparse level 1 routines describe operations between a vector in sparse format and a vector in dense format. This section describes all hipSPARSE level 1 sparse linear algebra functions.

1.8.1 hipsparseXaxpyi()

hipsparseStatus_t **hipsparseSaxpyi**(*hipsparseHandle_t* handle, int nnz, const float *alpha, const float *xVal, const int *xInd, float *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDaxpyi**(*hipsparseHandle_t* handle, int nnz, const double *alpha, const double *xVal, const int *xInd, double *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCaxpyi**(*hipsparseHandle_t* handle, int nnz, const hipComplex *alpha, const hipComplex *xVal, const int *xInd, hipComplex *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZaxpyi**(*hipsparseHandle_t* handle, int nnz, const hipDoubleComplex *alpha, const hipDoubleComplex *xVal, const int *xInd, hipDoubleComplex *y, *hipsparseIndexBase_t* idxBase)

Scale a sparse vector and add it to a dense vector.

`hipsparseXaxpyi` multiplies the sparse vector x with scalar α and adds the result to the dense vector y, such that

$$y := y + \alpha \cdot x$$

```
for(i = 0; i < nnz; ++i)
{
    y[x_ind[i]] = y[x_ind[i]] + alpha * x_val[i];
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.2 hipsparseXdoti()

hipsparseStatus_t **hipsparseSdoti**(*hipsparseHandle_t* handle, int nnz, const float *xVal, const int *xInd, const float *y, float *result, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDdoti**(*hipsparseHandle_t* handle, int nnz, const double *xVal, const int *xInd, const double *y, double *result, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCdoti**(*hipsparseHandle_t* handle, int nnz, const hipComplex *xVal, const int *xInd, const hipComplex *y, hipComplex *result, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZdoti**(*hipsparseHandle_t* handle, int nnz, const hipDoubleComplex *xVal, const int *xInd, const hipDoubleComplex *y, hipDoubleComplex *result, *hipsparseIndexBase_t* idxBase)

Compute the dot product of a sparse vector with a dense vector.

`hipsparseXdoti` computes the dot product of the sparse vector x with the dense vector y, such that

$$\text{result} := y^T x$$

```
for(i = 0; i < nnz; ++i)
{
    result += x_val[i] * y[x_ind[i]];
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.3 hipsparseXdotci()

hipsparseStatus_t **hipsparseCdotci**(*hipsparseHandle_t* handle, int nnz, const hipComplex *xVal, const int *xInd, const hipComplex *y, hipComplex *result, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZdotci**(*hipsparseHandle_t* handle, int nnz, const hipDoubleComplex *xVal, const int *xInd, const hipDoubleComplex *y, hipDoubleComplex *result, *hipsparseIndexBase_t* idxBase)

Compute the dot product of a complex conjugate sparse vector with a dense vector.

`hipsparseXdotci` computes the dot product of the complex conjugate sparse vector x with the dense vector y, such that

$$\text{result} := \bar{x}^H y$$

```
for(i = 0; i < nnz; ++i)
{
    result += conj(x_val[i]) * y[x_ind[i]];
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.4 hipsparseXgthr()

hipsparseStatus_t **hipsparseSgthr**(*hipsparseHandle_t* handle, int nnz, const float *y, float *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDgthr**(*hipsparseHandle_t* handle, int nnz, const double *y, double *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCgthr**(*hipsparseHandle_t* handle, int nnz, const hipComplex *y, hipComplex *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZgthr**(*hipsparseHandle_t* handle, int nnz, const hipDoubleComplex *y, hipDoubleComplex *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

Gather elements from a dense vector and store them into a sparse vector.

`hipsparseXgthr` gathers the elements that are listed in `x_ind` from the dense vector y and stores them in the sparse vector x.

```
for(i = 0; i < nnz; ++i)
{
    x_val[i] = y[x_ind[i]];
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.5 hipsparseXgthrz()

hipsparseStatus_t **hipsparseSgthrz**(*hipsparseHandle_t* handle, int nnz, float *y, float *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDgthrz**(*hipsparseHandle_t* handle, int nnz, double *y, double *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCgthrz**(*hipsparseHandle_t* handle, int nnz, hipComplex *y, hipComplex *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZgthrz**(*hipsparseHandle_t* handle, int nnz, hipDoubleComplex *y, hipDoubleComplex *xVal, const int *xInd, *hipsparseIndexBase_t* idxBase)

Gather and zero out elements from a dense vector and store them into a sparse vector.

`hipsparseXgthrz` gathers the elements that are listed in `x_ind` from the dense vector y and stores them in the sparse vector x. The gathered elements in y are replaced by zero.

```
for(i = 0; i < nnz; ++i)
{
    x_val[i]    = y[x_ind[i]];
    y[x_ind[i]] = 0;
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.6 hipsparseXroti()

hipsparseStatus_t **hipsparseSroti**(*hipsparseHandle_t* handle, int nnz, float *xVal, const int *xInd, float *y, const float *c, const float *s, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDroti**(*hipsparseHandle_t* handle, int nnz, double *xVal, const int *xInd, double *y, const double *c, const double *s, *hipsparseIndexBase_t* idxBase)

Apply Givens rotation to a dense and a sparse vector.

`hipsparseXroti` applies the Givens rotation matrix G to the sparse vector x and the dense vector y, where

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

```
for(i = 0; i < nnz; ++i)
{
    x_tmp = x_val[i];
    y_tmp = y[x_ind[i]];

    x_val[i]    = c * x_tmp + s * y_tmp;
    y[x_ind[i]] = c * y_tmp - s * x_tmp;
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.8.7 hipsparseXsctr()

hipsparseStatus_t **hipsparseSsctr**(*hipsparseHandle_t* handle, int nnz, const float *xVal, const int *xInd, float *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDsctr**(*hipsparseHandle_t* handle, int nnz, const double *xVal, const int *xInd, double *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCsctr**(*hipsparseHandle_t* handle, int nnz, const hipComplex *xVal, const int *xInd, hipComplex *y, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZsctr**(*hipsparseHandle_t* handle, int nnz, const hipDoubleComplex *xVal, const int *xInd, hipDoubleComplex *y, *hipsparseIndexBase_t* idxBase)

Scatter elements from a dense vector across a sparse vector.

`hipsparseXsctr` scatters the elements that are listed in `x_ind` from the sparse vector x into the dense vector y. Indices of y that are not listed in `x_ind` remain unchanged.

```
for(i = 0; i < nnz; ++i)
{
    y[x_ind[i]] = x_val[i];
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.9 Sparse Level 2 Functions

This module holds all sparse level 2 routines.

The sparse level 2 routines describe operations between a matrix in sparse format and a vector in dense format.

1.9.1 hipsparseXcsrmv()

hipsparseStatus_t **hipsparseScsrmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *x, const float *beta, float *y)

hipsparseStatus_t **hipsparseDcsrmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *x, const double *beta, double *y)

hipsparseStatus_t **hipsparseCcsrmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *x, const hipComplex *beta, hipComplex *y)

hipsparseStatus_t **hipsparseZcsrmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *x, const hipDoubleComplex *beta, hipDoubleComplex *y)

Sparse matrix vector multiplication using CSR storage format.

`hipsparseXcsrmv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in CSR storage format, and the dense vector x and adds the result to the dense vector y that is multiplied by the scalar β, such that

$$y := \alpha \cdot op(A) \cdot x + \beta \cdot y,$$

with

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

```
for(i = 0; i < m; ++i)
{
    y[i] = beta * y[i];

    for(j = csr_row_ptr[i]; j < csr_row_ptr[i + 1]; ++j)
    {
        y[i] = y[i] + alpha * csr_val[j] * x[csr_col_ind[j]];
    }
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

1.9.2 hipsparseXcsrsv2_zeroPivot()

hipsparseStatus_t **hipsparseXcsrsv2_zeroPivot**(*hipsparseHandle_t* handle, *csrsv2Info_t* info, int *position)

Sparse triangular solve using CSR storage format.

`hipsparseXcsrsv2_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during *hipsparseScsrsv2_solve()*, *hipsparseDcsrsv2_solve()*, *hipsparseCcsrsv2_solve()* or *hipsparseZcsrsv2_solve()* computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the CSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXcsrsv2_zeroPivot` is a blocking function. It might influence performance negatively.

1.9.3 hipsparseXcsrsv2_bufferSize()

hipsparseStatus_t **hipsparseScsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, int *pBufferSizeInBytes)

Sparse triangular solve using CSR storage format.

`hipsparseXcsrsv2_bufferSize` returns the size of the temporary storage buffer that is required by *hipsparseScsrsv2_analysis()*, *hipsparseDcsrsv2_analysis()*, *hipsparseCcsrsv2_analysis()*, *hipsparseZcsrsv2_analysis()*, *hipsparseScsrsv2_solve()*, *hipsparseDcsrsv2_solve()*, *hipsparseCcsrsv2_solve()* and *hipsparseZcsrsv2_solve()*. The temporary storage buffer must be allocated by the user.

1.9.4 hipsparseXcsrsv2_bufferSizeExt()

hipsparseStatus_t **hipsparseScsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseDcsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseCcsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseZcsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, size_t *pBufferSize)

Sparse triangular solve using CSR storage format.

`hipsparseXcsrsv2_bufferSizeExt` returns the size of the temporary storage buffer that is required by *hipsparseScsrsv2_analysis()*, *hipsparseDcsrsv2_analysis()*, *hipsparseCcsrsv2_analysis()*, *hipsparseZcsrsv2_analysis()*, *hipsparseScsrsv2_solve()*, *hipsparseDcsrsv2_solve()*, *hipsparseCcsrsv2_solve()* and *hipsparseZcsrsv2_solve()*. The temporary storage buffer must be allocated by the user.

1.9.5 hipsparseXcsrsv2_analysis()

hipsparseStatus_t **hipsparseScsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular solve using CSR storage format.

`hipsparseXcsrsv2_analysis` performs the analysis step for *hipsparseScsrsv2_solve()*, *hipsparseDcsrsv2_solve()*, *hipsparseCcsrsv2_solve()* and *hipsparseZcsrsv2_solve()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.9.6 hipsparseXcsrsv2_solve()

hipsparseStatus_t **hipsparseScsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, const float *f, float *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, const double *f, double *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, const hipComplex *f, hipComplex *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrsv2Info_t* info, const hipDoubleComplex *f, hipDoubleComplex *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular solve using CSR storage format.

`hipsparseXcsrsv2_solve` solves a sparse triangular linear system of a sparse $m \times m$ matrix, defined in CSR storage format, a dense solution vector y and the right-hand side x that is multiplied by α, such that

$$op(A) \cdot y = \alpha \cdot x,$$

with

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

`hipsparseXcsrsv2_solve` requires a user allocated temporary buffer. Its size is returned by hipsparseXcsrsv2_bufferSize() or hipsparseXcsrsv2_bufferSizeExt(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXcsrsv2_analysis(). `hipsparseXcsrsv2_solve` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be checked calling *hipsparseXcsrsv2_zeroPivot()*. If *hipsparseDiagType_t* == HIPSPARSE_DIAG_TYPE_UNIT, no zero pivot will be reported, even if $A_{j,j} = 0$ for some j.

Note: The sparse CSR matrix has to be sorted. This can be achieved by calling *hipsparseXcsrsort()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans == HIPSPARSE_OPERATION_NON_TRANSPOSE` and `trans == HIPSPARSE_OPERATION_TRANSPOSE` is supported.

1.9.7 hipsparseXhybmv()

hipsparseStatus_t **hipsparseShybmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, const float *alpha, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, const float *x, const float *beta, float *y)

hipsparseStatus_t **hipsparseDhybmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, const double *alpha, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, const double *x, const double *beta, double *y)

hipsparseStatus_t **hipsparseChybmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, const hipComplex *x, const hipComplex *beta, hipComplex *y)

hipsparseStatus_t **hipsparseZhybmv**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, const hipDoubleComplex *x, const hipDoubleComplex *beta, hipDoubleComplex *y)

Sparse matrix vector multiplication using HYB storage format.

`hipsparseXhybmv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in HYB storage format, and the dense vector x and adds the result to the dense vector y that is multiplied by the scalar β, such that

$$y := \alpha \cdot op(A) \cdot x + \beta \cdot y,$$

with

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

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans == HIPSPARSE_OPERATION_NON_TRANSPOSE` is supported.

1.9.8 hipsparseXbsrmv()

hipsparseStatus_t **hipsparseSbsrmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nb, int nnzb, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, const float *x, const float *beta, float *y)

hipsparseStatus_t **hipsparseDbsrmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nb, int nnzb, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, const double *x, const double *beta, double *y)

hipsparseStatus_t **hipsparseCbsrmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nb, int nnzb, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, const hipComplex *x, const hipComplex *beta, hipComplex *y)

hipsparseStatus_t **hipsparseZbsrmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nb, int nnzb, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, const hipDoubleComplex *x, const hipDoubleComplex *beta, hipDoubleComplex *y)

Sparse matrix vector multiplication using BSR storage format.

`hipsparseXbsrmv` multiplies the scalar α with a sparse $(mb \cdot \text{block\_dim}) \times (nb \cdot \text{block\_dim})$ matrix, defined in BSR storage format, and the dense vector x and adds the result to the dense vector y that is multiplied by the scalar β, such that

$$y := \alpha \cdot op(A) \cdot x + \beta \cdot y,$$

with

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

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

1.9.9 hipsparseXbsrxmv()

hipsparseStatus_t **hipsparseSbsrxmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, *hipsparseOperation_t* trans, int sizeOfMask, int mb, int nb, int nnzb, const float *alpha, const *hipsparseMatDescr_t* descr, const float *bsrVal, const int *bsrMaskPtr, const int *bsrRowPtr, const int *bsrEndPtr, const int *bsrColInd, int blockDim, const float *x, const float *beta, float *y)

hipsparseStatus_t **hipsparseDbsrxmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, *hipsparseOperation_t* trans, int sizeOfMask, int mb, int nb, int nnzb, const double *alpha, const *hipsparseMatDescr_t* descr, const double *bsrVal, const int *bsrMaskPtr, const int *bsrRowPtr, const int *bsrEndPtr, const int *bsrColInd, int blockDim, const double *x, const double *beta, double *y)

hipsparseStatus_t **hipsparseCbsrxmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, *hipsparseOperation_t* trans, int sizeOfMask, int mb, int nb, int nnzb, const hipComplex *alpha, const *hipsparseMatDescr_t* descr, const hipComplex *bsrVal, const int *bsrMaskPtr, const int *bsrRowPtr, const int *bsrEndPtr, const int *bsrColInd, int blockDim, const hipComplex *x, const hipComplex *beta, hipComplex *y)

hipsparseStatus_t **hipsparseZbsrxmv**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, *hipsparseOperation_t* trans, int sizeOfMask, int mb, int nb, int nnzb, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descr, const hipDoubleComplex *bsrVal, const int *bsrMaskPtr, const int *bsrRowPtr, const int *bsrEndPtr, const int *bsrColInd, int blockDim, const hipDoubleComplex *x, const hipDoubleComplex *beta, hipDoubleComplex *y)

Sparse matrix vector multiplication with mask operation using BSR storage format.

`hipsparseXbsrxmv` multiplies the scalar α with a sparse $(mb \cdot \text{block\_dim}) \times (nb \cdot \text{block\_dim})$ modified matrix, defined in BSR storage format, and the dense vector x and adds the result to the dense vector y that is multiplied by the scalar β, such that

$$y := (\alpha \cdot op(A) \cdot x + \beta \cdot y) \, (\text{mask}) \, ,$$

with

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

The mask is defined as an array of block row indices. The input sparse matrix is defined with a modified BSR storage format where the beginning and the end of each row is defined with two arrays, `bsr_row_ptr` and `bsr_end_ptr` (both of size `mb`), rather the usual `bsr_row_ptr` of size `mb` + 1.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported. Currently, `block_dim` == 1 is not supported.

1.9.10 hipsparseXbsrsv2_zeroPivot()

hipsparseStatus_t **hipsparseXbsrsv2_zeroPivot**(*hipsparseHandle_t* handle, *bsrsv2Info_t* info, int *position)

Sparse triangular solve using BSR storage format.

`hipsparseXbsrsv2_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXbsrsv2_analysis() or hipsparseXbsrsv2_solve() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the BSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXbsrsv2_zeroPivot` is a blocking function. It might influence performance negatively.

1.9.11 hipsparseXbsrsv2_bufferSize()

hipsparseStatus_t **hipsparseSbsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDbsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCbsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZbsrsv2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, int *pBufferSizeInBytes)

Sparse triangular solve using BSR storage format.

`hipsparseXbsrsv2_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXbsrsv2_analysis() and hipsparseXbsrsv2_solve(). The temporary storage buffer must be allocated by the user.

1.9.12 hipsparseXbsrsv2_bufferSizeExt()

hipsparseStatus_t **hipsparseSbsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseDbsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseCbsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseZbsrsv2_bufferSizeExt**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, size_t *pBufferSize)

Sparse triangular solve using BSR storage format.

`hipsparseXbsrsv2_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXbsrsv2_analysis() and hipsparseXbsrsv2_solve(). The temporary storage buffer must be allocated by the user.

1.9.13 hipsparseXbsrsv2_analysis()

hipsparseStatus_t **hipsparseSbsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrsv2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular solve using BSR storage format.

`hipsparseXbsrsv2_analysis` performs the analysis step for hipsparseXbsrsv2_solve().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.9.14 hipsparseXbsrsv2_solve()

hipsparseStatus_t **hipsparseSbsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, const float *f, float *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, const double *f, double *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, const hipComplex *f, hipComplex *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrsv2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, int mb, int nnzb, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsv2Info_t* info, const hipDoubleComplex *f, hipDoubleComplex *x, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular solve using BSR storage format.

`hipsparseXbsrsv2_solve` solves a sparse triangular linear system of a sparse $m \times m$ matrix, defined in BSR storage format, a dense solution vector y and the right-hand side x that is multiplied by α, such that

$$op(A) \cdot y = \alpha \cdot x,$$

with

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

`hipsparseXbsrsv2_solve` requires a user allocated temporary buffer. Its size is returned by hipsparseXbsrsv2_bufferSize() or hipsparseXbsrsv2_bufferSizeExt(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXbsrsv2_analysis(). `hipsparseXbsrsv2_solve` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be checked calling *hipsparseXbsrsv2_zeroPivot()*. If *hipsparseDiagType_t* == HIPSPARSE_DIAG_TYPE_UNIT, no zero pivot will be reported, even if $A_{j,j} = 0$ for some j.

Note: The sparse BSR matrix has to be sorted.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans == HIPSPARSE_OPERATION_NON_TRANSPOSE` and `trans == HIPSPARSE_OPERATION_TRANSPOSE` is supported.

1.9.15 hipsparseXgemvi_bufferSize()

hipsparseStatus_t **hipsparseSgemvi_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, int *pBufferSize)

hipsparseStatus_t **hipsparseDgemvi_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, int *pBufferSize)

hipsparseStatus_t **hipsparseCgemvi_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, int *pBufferSize)

hipsparseStatus_t **hipsparseZgemvi_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int nnz, int *pBufferSize)

Dense matrix sparse vector multiplication.

`hipsparseXgemvi_bufferSize` returns the size of the temporary storage buffer required by hipsparseXgemvi(). The temporary storage buffer must be allocated by the user.

1.9.16 hipsparseXgemvi()

hipsparseStatus_t **hipsparseSgemvi**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, const float *alpha, const float *A, int lda, int nnz, const float *x, const int *xInd, const float *beta, float *y, *hipsparseIndexBase_t* idxBase, void *pBuffer)

hipsparseStatus_t **hipsparseDgemvi**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, const double *alpha, const double *A, int lda, int nnz, const double *x, const int *xInd, const double *beta, double *y, *hipsparseIndexBase_t* idxBase, void *pBuffer)

hipsparseStatus_t **hipsparseCgemvi**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, const hipComplex *alpha, const hipComplex *A, int lda, int nnz, const hipComplex *x, const int *xInd, const hipComplex *beta, hipComplex *y, *hipsparseIndexBase_t* idxBase, void *pBuffer)

hipsparseStatus_t **hipsparseZgemvi**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, const hipDoubleComplex *alpha, const hipDoubleComplex *A, int lda, int nnz, const hipDoubleComplex *x, const int *xInd, const hipDoubleComplex *beta, hipDoubleComplex *y, *hipsparseIndexBase_t* idxBase, void *pBuffer)

Dense matrix sparse vector multiplication.

`hipsparseXgemvi` multiplies the scalar α with a dense $m \times n$ matrix A and the sparse vector x and adds the result to the dense vector y that is multiplied by the scalar β, such that

$$y := \alpha \cdot op(A) \cdot x + \beta \cdot y,$$

with

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

`hipsparseXgemvi` requires a user allocated temporary buffer. Its size is returned by hipsparseXgemvi_bufferSize().

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

1.10 Sparse Level 3 Functions

This module holds all sparse level 3 routines.

The sparse level 3 routines describe operations between a matrix in sparse format and multiple vectors in dense format that can also be seen as a dense matrix.

1.10.1 hipsparseXbsrmm()

hipsparseStatus_t **hipsparseSbsrmm**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int mb, int n, int kb, int nnzb, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const float *B, int ldb, const float *beta, float *C, int ldc)

hipsparseStatus_t **hipsparseDbsrmm**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int mb, int n, int kb, int nnzb, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const double *B, int ldb, const double *beta, double *C, int ldc)

hipsparseStatus_t **hipsparseCbsrmm**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int mb, int n, int kb, int nnzb, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const hipComplex *B, int ldb, const hipComplex *beta, hipComplex *C, int ldc)

hipsparseStatus_t **hipsparseZbsrmm**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int mb, int n, int kb, int nnzb, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const hipDoubleComplex *B, int ldb, const hipDoubleComplex *beta, hipDoubleComplex *C, int ldc)

Sparse matrix dense matrix multiplication using BSR storage format.

`hipsparseXbsrmm` multiplies the scalar α with a sparse $mb \times kb$ matrix A, defined in BSR storage format, and the dense $k \times n$ matrix B (where $k = block\_dim \times kb$) and adds the result to the dense $m \times n$ matrix C (where $m = block\_dim \times mb$) that is multiplied by the scalar β, such that

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C,$$

with

$$op(A) = \begin{cases} A, & \text{if trans\_A == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \end{cases}$$

and

$$op(B) = \begin{cases} B, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \end{cases}$$

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans_A` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

1.10.2 hipsparseXcsrmm()

hipsparseStatus_t **hipsparseScsrmm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int k, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *B, int ldb, const float *beta, float *C, int ldc)

hipsparseStatus_t **hipsparseDcsrmm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int k, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *B, int ldb, const double *beta, double *C, int ldc)

hipsparseStatus_t **hipsparseCcsrmm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int k, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *B, int ldb, const hipComplex *beta, hipComplex *C, int ldc)

hipsparseStatus_t **hipsparseZcsrmm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, int m, int n, int k, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *B, int ldb, const hipDoubleComplex *beta, hipDoubleComplex *C, int ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`hipsparseXcsrmm` multiplies the scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and the dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by the scalar β, such that

$$C := \alpha \cdot op(A) \cdot B + \beta \cdot C,$$

with

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

```
for(i = 0; i < ldc; ++i)
{
    for(j = 0; j < n; ++j)
    {
        C[i][j] = beta * C[i][j];

        for(k = csr_row_ptr[i]; k < csr_row_ptr[i + 1]; ++k)
        {
            C[i][j] += alpha * csr_val[k] * B[csr_col_ind[k]][j];
        }
    }
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.10.3 hipsparseXcsrmm2()

hipsparseStatus_t **hipsparseScsrmm2**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *B, int ldb, const float *beta, float *C, int ldc)

hipsparseStatus_t **hipsparseDcsrmm2**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *B, int ldb, const double *beta, double *C, int ldc)

hipsparseStatus_t **hipsparseCcsrmm2**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *B, int ldb, const hipComplex *beta, hipComplex *C, int ldc)

hipsparseStatus_t **hipsparseZcsrmm2**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *B, int ldb, const hipDoubleComplex *beta, hipDoubleComplex *C, int ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`hipsparseXcsrmm2` multiplies the scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and the dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by the scalar β, such that

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C,$$

with

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

and

$$op(B) = \begin{cases} B, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ B^H, & \text{if trans\_B == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

```
for(i = 0; i < ldc; ++i)
{
    for(j = 0; j < n; ++j)
    {
        C[i][j] = beta * C[i][j];

        for(k = csr_row_ptr[i]; k < csr_row_ptr[i + 1]; ++k)
        {
            C[i][j] += alpha * csr_val[k] * B[csr_col_ind[k]][j];
        }
    }
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.10.4 hipsparseXbsrsm2_zeroPivot()

hipsparseStatus_t **hipsparseXbsrsm2_zeroPivot**(*hipsparseHandle_t* handle, *bsrsm2Info_t* info, int *position)

Sparse triangular system solve using BSR storage format.

`hipsparseXbsrsm2_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXbsrsm2_analysis() or hipsparseXbsrsm2_solve() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the BSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXbsrsm2_zeroPivot` is a blocking function. It might influence performance negatively.

1.10.5 hipsparseXbsrsm2_bufferSize()

hipsparseStatus_t **hipsparseSbsrsm2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDbsrsm2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCbsrsm2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZbsrsm2_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, int *pBufferSizeInBytes)

Sparse triangular system solve using BSR storage format.

`hipsparseXbsrsm2_buffer_size` returns the size of the temporary storage buffer that is required by hipsparseXbsrsm2_analysis() and hipsparseXbsrsm2_solve(). The temporary storage buffer must be allocated by the user.

1.10.6 hipsparseXbsrsm2_analysis()

hipsparseStatus_t **hipsparseSbsrsm2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, const float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrsm2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, const double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrsm2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrsm2_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular system solve using BSR storage format.

`hipsparseXbsrsm2_analysis` performs the analysis step for hipsparseXbsrsm2_solve().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.10.7 hipsparseXbsrsm2_solve()

hipsparseStatus_t **hipsparseSbsrsm2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, const float *B, int ldb, float *X, int ldx, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrsm2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, const double *B, int ldb, double *X, int ldx, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrsm2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, const hipComplex *B, int ldb, hipComplex *X, int ldx, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrsm2_solve**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, *hipsparseOperation_t* transA, *hipsparseOperation_t* transX, int mb, int nrhs, int nnzb, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrsm2Info_t* info, const hipDoubleComplex *B, int ldb, hipDoubleComplex *X, int ldx, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular system solve using BSR storage format.

`hipsparseXbsrsm2_solve` solves a sparse triangular linear system of a sparse $m \times m$ matrix, defined in BSR storage format, a dense solution matrix X and the right-hand side matrix B that is multiplied by α, such that

$$op(A) \cdot op(X) = \alpha \cdot op(B),$$

with

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

,

$$op(X) = \begin{cases} X, & \text{if trans\_X == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T, & \text{if trans\_X == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ X^H, & \text{if trans\_X == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

`hipsparseXbsrsm2_solve` requires a user allocated temporary buffer. Its size is returned by hipsparseXbsrsm2_bufferSize(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXbsrsm2_analysis(). `hipsparseXbsrsm2_solve` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be checked calling *hipsparseXbsrsm2_zeroPivot()*. If *hipsparseDiagType_t* == HIPSPARSE_DIAG_TYPE_UNIT, no zero pivot will be reported, even if $A_{j,j} = 0$ for some j.

Note: The sparse BSR matrix has to be sorted.

Note: Operation type of B and X must match, if $op(B) = B, op(X) = X$.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans_A` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE and `trans_X` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE is supported.

1.10.8 hipsparseXcsrsm2_zeroPivot()

hipsparseStatus_t **hipsparseXcsrsm2_zeroPivot**(*hipsparseHandle_t* handle, *csrsm2Info_t* info, int *position)

Sparse triangular system solve using CSR storage format.

`hipsparseXcsrsm2_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXcsrsm2_analysis() or hipsparseXcsrsm2_solve() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the CSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXcsrsm2_zeroPivot` is a blocking function. It might influence performance negatively.

1.10.9 hipsparseXcsrsm2_bufferSizeExt()

hipsparseStatus_t **hipsparseScsrsm2_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, size_t *pBufferSize)

hipsparseStatus_t **hipsparseDcsrsm2_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, size_t *pBufferSize)

hipsparseStatus_t **hipsparseCcsrsm2_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, size_t *pBufferSize)

hipsparseStatus_t **hipsparseZcsrsm2_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, size_t *pBufferSize)

Sparse triangular system solve using BSR storage format.

`hipsparseXbsrsm2_solve` solves a sparse triangular linear system of a sparse $m \times m$ matrix, defined in BSR storage format, a dense solution matrix X and the right-hand side matrix B that is multiplied by α, such that

$$op(A) \cdot op(X) = \alpha \cdot op(B),$$

with

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

,

$$op(X) = \begin{cases} X, & \text{if trans\_X == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T, & \text{if trans\_X == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ X^H, & \text{if trans\_X == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

`hipsparseXbsrsm2_solve` requires a user allocated temporary buffer. Its size is returned by hipsparseXbsrsm2_bufferSize(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXbsrsm2_analysis(). `hipsparseXbsrsm2_solve` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be checked calling *hipsparseXbsrsm2_zeroPivot()*. If *hipsparseDiagType_t* == HIPSPARSE_DIAG_TYPE_UNIT, no zero pivot will be reported, even if $A_{j,j} = 0$ for some j.

Note: The sparse BSR matrix has to be sorted.

Note: Operation type of B and X must match, if $op(B) = B, op(X) = X$.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans_A` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE and `trans_X` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE is supported.

1.10.10 hipsparseXcsrsm2_analysis()

hipsparseStatus_t **hipsparseScsrsm2_analysis**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrsm2_analysis**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrsm2_analysis**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrsm2_analysis**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular system solve using CSR storage format.

`hipsparseXcsrsm2_analysis` performs the analysis step for hipsparseXcsrsm2_solve().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.10.11 hipsparseXcsrsm2_solve()

hipsparseStatus_t **hipsparseScsrsm2_solve**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const float *alpha, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, float *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrsm2_solve**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const double *alpha, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, double *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrsm2_solve**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, hipComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrsm2_solve**(*hipsparseHandle_t* handle, int algo, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int nrhs, int nnz, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, hipDoubleComplex *B, int ldb, *csrsm2Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Sparse triangular system solve using CSR storage format.

`hipsparseXcsrsm2_solve` solves a sparse triangular linear system of a sparse $m \times m$ matrix, defined in CSR storage format, a dense solution matrix X and the right-hand side matrix B that is multiplied by α, such that

$$op(A) \cdot op(X) = \alpha \cdot op(B),$$

with

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

,

$$op(B) = \begin{cases} B, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ B^H, & \text{if trans\_B == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$op(X) = \begin{cases} X, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ X^H, & \text{if trans\_B == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

`hipsparseXcsrsm2_solve` requires a user allocated temporary buffer. Its size is returned by hipsparseXcsrsm2_bufferSizeExt(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXcsrsm2_analysis(). `hipsparseXcsrsm2_solve` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be checked calling *hipsparseXcsrsm2_zeroPivot()*. If *hipsparseDiagType_t* == HIPSPARSE_DIAG_TYPE_UNIT, no zero pivot will be reported, even if $A_{j,j} = 0$ for some j.

Note: The sparse CSR matrix has to be sorted. This can be achieved by calling *hipsparseXcsrsort()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only `trans_A` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE and `trans_B` != HIPSPARSE_OPERATION_CONJUGATE_TRANSPOSE is supported.

1.10.12 hipsparseXgemmi()

hipsparseStatus_t **hipsparseSgemmi**(*hipsparseHandle_t* handle, int m, int n, int k, int nnz, const float *alpha, const float *A, int lda, const float *cscValB, const int *cscColPtrB, const int *cscRowIndB, const float *beta, float *C, int ldc)

hipsparseStatus_t **hipsparseDgemmi**(*hipsparseHandle_t* handle, int m, int n, int k, int nnz, const double *alpha, const double *A, int lda, const double *cscValB, const int *cscColPtrB, const int *cscRowIndB, const double *beta, double *C, int ldc)

hipsparseStatus_t **hipsparseCgemmi**(*hipsparseHandle_t* handle, int m, int n, int k, int nnz, const hipComplex *alpha, const hipComplex *A, int lda, const hipComplex *cscValB, const int *cscColPtrB, const int *cscRowIndB, const hipComplex *beta, hipComplex *C, int ldc)

hipsparseStatus_t **hipsparseZgemmi**(*hipsparseHandle_t* handle, int m, int n, int k, int nnz, const hipDoubleComplex *alpha, const hipDoubleComplex *A, int lda, const hipDoubleComplex *cscValB, const int *cscColPtrB, const int *cscRowIndB, const hipDoubleComplex *beta, hipDoubleComplex *C, int ldc)

Dense matrix sparse matrix multiplication using CSR storage format.

`hipsparseXgemmi` multiplies the scalar α with a dense $m \times k$ matrix A and the sparse $k \times n$ matrix B, defined in CSR storage format and adds the result to the dense $m \times n$ matrix C that is multiplied by the scalar β, such that

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

with

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

and

$$op(B) = \begin{cases} B, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ B^H, & \text{if trans\_B == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.11 Sparse Extra Functions

This module holds all sparse extra routines.

The sparse extra routines describe operations that manipulate sparse matrices.

1.11.1 hipsparseXcsrgeamNnz()

hipsparseStatus_t **hipsparseXcsrgeamNnz**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr)

Sparse matrix sparse matrix addition using CSR storage format.

`hipsparseXcsrgeamNnz` computes the total CSR non-zero elements and the CSR row offsets, that point to the start of every row of the sparse CSR matrix, of the resulting matrix C. It is assumed that `csr_row_ptr_C` has been allocated with size `m` + 1.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.2 hipsparseXcsrgeam()

hipsparseStatus_t **hipsparseScsrgeam**(*hipsparseHandle_t* handle, int m, int n, const float *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const float *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, float *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseDcsrgeam**(*hipsparseHandle_t* handle, int m, int n, const double *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const double *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, double *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseCcsrgeam**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const hipComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, hipComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseZcsrgeam**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const hipDoubleComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipDoubleComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

Sparse matrix sparse matrix addition using CSR storage format.

`hipsparseXcsrgeam` multiplies the scalar α with the sparse $m \times n$ matrix A, defined in CSR storage format, multiplies the scalar β with the sparse $m \times n$ matrix B, defined in CSR storage format, and adds both resulting matrices to obtain the sparse $m \times n$ matrix C, defined in CSR storage format, such that

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

It is assumed that `csr_row_ptr_C` has already been filled and that `csr_val_C` and `csr_col_ind_C` are allocated by the user. `csr_row_ptr_C` and allocation size of `csr_col_ind_C` and `csr_val_C` is defined by the number of non-zero elements of the sparse CSR matrix C. Both can be obtained by *hipsparseXcsrgeamNnz()*.

Note: Both scalars α and $beta$ have to be valid.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.11.3 hipsparseXcsrgeam2_bufferSizeExt()

hipsparseStatus_t **hipsparseScsrgeam2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const float *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const float *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, const float *csrSortedValC, const int *csrSortedRowPtrC, const int *csrSortedColIndC, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsrgeam2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const double *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const double *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, const double *csrSortedValC, const int *csrSortedRowPtrC, const int *csrSortedColIndC, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsrgeam2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipComplex *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, const hipComplex *csrSortedValC, const int *csrSortedRowPtrC, const int *csrSortedColIndC, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsrgeam2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipDoubleComplex *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, const hipDoubleComplex *csrSortedValC, const int *csrSortedRowPtrC, const int *csrSortedColIndC, size_t *pBufferSizeInBytes)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgeam2_bufferSizeExt` returns the size of the temporary storage buffer that is required by *hipsparseXcsrgeam2Nnz()* and hipsparseXcsrgeam2(). The temporary storage buffer must be allocated by the user.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.4 hipsparseXcsrgeam2Nnz()

hipsparseStatus_t **hipsparseXcsrgeam2Nnz**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, int *csrSortedRowPtrC, int *nnzTotalDevHostPtr, void *workspace)

Sparse matrix sparse matrix addition using CSR storage format.

`hipsparseXcsrgeam2Nnz` computes the total CSR non-zero elements and the CSR row offsets, that point to the start of every row of the sparse CSR matrix, of the resulting matrix C. It is assumed that `csr_row_ptr_C` has been allocated with size `m` + 1.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.5 hipsparseXcsrgeam2()

hipsparseStatus_t **hipsparseScsrgeam2**(*hipsparseHandle_t* handle, int m, int n, const float *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const float *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const float *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, float *csrSortedValC, int *csrSortedRowPtrC, int *csrSortedColIndC, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrgeam2**(*hipsparseHandle_t* handle, int m, int n, const double *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const double *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const double *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, double *csrSortedValC, int *csrSortedRowPtrC, int *csrSortedColIndC, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrgeam2**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipComplex *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, hipComplex *csrSortedValC, int *csrSortedRowPtrC, int *csrSortedColIndC, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrgeam2**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, const hipDoubleComplex *beta, const *hipsparseMatDescr_t* descrB, int nnzB, const hipDoubleComplex *csrSortedValB, const int *csrSortedRowPtrB, const int *csrSortedColIndB, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrSortedValC, int *csrSortedRowPtrC, int *csrSortedColIndC, void *pBuffer)

Sparse matrix sparse matrix addition using CSR storage format.

`hipsparseXcsrgeam2` multiplies the scalar α with the sparse $m \times n$ matrix A, defined in CSR storage format, multiplies the scalar β with the sparse $m \times n$ matrix B, defined in CSR storage format, and adds both resulting matrices to obtain the sparse $m \times n$ matrix C, defined in CSR storage format, such that

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

It is assumed that `csr_row_ptr_C` has already been filled and that `csr_val_C` and `csr_col_ind_C` are allocated by the user. `csr_row_ptr_C` and allocation size of `csr_col_ind_C` and `csr_val_C` is defined by the number of non-zero elements of the sparse CSR matrix C. Both can be obtained by *hipsparseXcsrgeam2Nnz()*.

Note: Both scalars α and $beta$ have to be valid.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.11.6 hipsparseXcsrgemmNnz()

hipsparseStatus_t **hipsparseXcsrgemmNnz**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgemmNnz` computes the total CSR non-zero elements and the CSR row offsets, that point to the start of every row of the sparse CSR matrix, of the resulting multiplied matrix C. It is assumed that `csr_row_ptr_C` has been allocated with size `m` + 1.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Please note, that for matrix products with more than 8192 intermediate products per row, additional temporary storage buffer is allocated by the algorithm.

Note: Currently, only `trans_A` == `trans_B` == HIPSPARSE_OPERATION_NONE is supported.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.7 hipsparseXcsrgemm()

hipsparseStatus_t **hipsparseScsrgemm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const float *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, float *csrValC, const int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseDcsrgemm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const double *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, double *csrValC, const int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseCcsrgemm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const hipComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const hipComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, hipComplex *csrValC, const int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseZcsrgemm**(*hipsparseHandle_t* handle, *hipsparseOperation_t* transA, *hipsparseOperation_t* transB, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const hipDoubleComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrValC, const int *csrRowPtrC, int *csrColIndC)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgemm` multiplies the sparse $m \times k$ matrix A, defined in CSR storage format with the sparse $k \times n$ matrix B, defined in CSR storage format, and stores the result in the sparse $m \times n$ matrix C, defined in CSR storage format, such that

$$C := op(A) \cdot op(B),$$

with

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

and

$$op(B) = \begin{cases} B, & \text{if trans\_B == HIPSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T, & \text{if trans\_B == HIPSPARSE\_OPERATION\_TRANSPOSE} \\ B^H, & \text{if trans\_B == HIPSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is assumed that `csr_row_ptr_C` has already been filled and that `csr_val_C` and `csr_col_ind_C` are allocated by the user. `csr_row_ptr_C` and allocation size of `csr_col_ind_C` and `csr_val_C` is defined by the number of non-zero elements of the sparse CSR matrix C. Both can be obtained by *hipsparseXcsrgemmNnz()*.

Note: Currently, only `trans_A` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

Note: Currently, only `trans_B` == HIPSPARSE_OPERATION_NON_TRANSPOSE is supported.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Please note, that for matrix products with more than 4096 non-zero entries per row, additional temporary storage buffer is allocated by the algorithm.

1.11.8 hipsparseXcsrgemm2_bufferSizeExt()

hipsparseStatus_t **hipsparseScsrgemm2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int k, const float *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const float *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const int *csrRowPtrD, const int *csrColIndD, *csrgemm2Info_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsrgemm2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int k, const double *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const double *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const int *csrRowPtrD, const int *csrColIndD, *csrgemm2Info_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsrgemm2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int k, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const hipComplex *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const int *csrRowPtrD, const int *csrColIndD, *csrgemm2Info_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsrgemm2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int k, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const hipDoubleComplex *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const int *csrRowPtrD, const int *csrColIndD, *csrgemm2Info_t* info, size_t *pBufferSizeInBytes)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgemm2_bufferSizeExt` returns the size of the temporary storage buffer that is required by *hipsparseXcsrgemm2Nnz()* and hipsparseXcsrgemm2(). The temporary storage buffer must be allocated by the user.

Note: Please note, that for matrix products with more than 4096 non-zero entries per row, additional temporary storage buffer is allocated by the algorithm.

Note: Please note, that for matrix products with more than 8192 intermediate products per row, additional temporary storage buffer is allocated by the algorithm.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.9 hipsparseXcsrgemm2Nnz()

hipsparseStatus_t **hipsparseXcsrgemm2Nnz**(*hipsparseHandle_t* handle, int m, int n, int k, const *hipsparseMatDescr_t* descrA, int nnzA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const int *csrRowPtrB, const int *csrColIndB, const *hipsparseMatDescr_t* descrD, int nnzD, const int *csrRowPtrD, const int *csrColIndD, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr, const *csrgemm2Info_t* info, void *pBuffer)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgemm2Nnz` computes the total CSR non-zero elements and the CSR row offsets, that point to the start of every row of the sparse CSR matrix, of the resulting multiplied matrix C. It is assumed that `csr_row_ptr_C` has been allocated with size `m` + 1. The required buffer size can be obtained by hipsparseXcsrgemm2_bufferSizeExt().

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Please note, that for matrix products with more than 8192 intermediate products per row, additional temporary storage buffer is allocated by the algorithm.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

1.11.10 hipsparseXcsrgemm2()

hipsparseStatus_t **hipsparseScsrgemm2**(*hipsparseHandle_t* handle, int m, int n, int k, const float *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const float *csrValB, const int *csrRowPtrB, const int *csrColIndB, const float *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const float *csrValD, const int *csrRowPtrD, const int *csrColIndD, const *hipsparseMatDescr_t* descrC, float *csrValC, const int *csrRowPtrC, int *csrColIndC, const *csrgemm2Info_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrgemm2**(*hipsparseHandle_t* handle, int m, int n, int k, const double *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const double *csrValB, const int *csrRowPtrB, const int *csrColIndB, const double *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const double *csrValD, const int *csrRowPtrD, const int *csrColIndD, const *hipsparseMatDescr_t* descrC, double *csrValC, const int *csrRowPtrC, int *csrColIndC, const *csrgemm2Info_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrgemm2**(*hipsparseHandle_t* handle, int m, int n, int k, const hipComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const hipComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const hipComplex *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const hipComplex *csrValD, const int *csrRowPtrD, const int *csrColIndD, const *hipsparseMatDescr_t* descrC, hipComplex *csrValC, const int *csrRowPtrC, int *csrColIndC, const *csrgemm2Info_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrgemm2**(*hipsparseHandle_t* handle, int m, int n, int k, const hipDoubleComplex *alpha, const *hipsparseMatDescr_t* descrA, int nnzA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const *hipsparseMatDescr_t* descrB, int nnzB, const hipDoubleComplex *csrValB, const int *csrRowPtrB, const int *csrColIndB, const hipDoubleComplex *beta, const *hipsparseMatDescr_t* descrD, int nnzD, const hipDoubleComplex *csrValD, const int *csrRowPtrD, const int *csrColIndD, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrValC, const int *csrRowPtrC, int *csrColIndC, const *csrgemm2Info_t* info, void *pBuffer)

Sparse matrix sparse matrix multiplication using CSR storage format.

`hipsparseXcsrgemm2` multiplies the scalar α with the sparse $m \times k$ matrix A, defined in CSR storage format, and the sparse $k \times n$ matrix B, defined in CSR storage format, and adds the result to the sparse $m \times n$ matrix D that is multiplied by β. The final result is stored in the sparse $m \times n$ matrix C, defined in CSR storage format, such that

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

It is assumed that `csr_row_ptr_C` has already been filled and that `csr_val_C` and `csr_col_ind_C` are allocated by the user. `csr_row_ptr_C` and allocation size of `csr_col_ind_C` and `csr_val_C` is defined by the number of non-zero elements of the sparse CSR matrix C. Both can be obtained by *hipsparseXcsrgemm2Nnz()*. The required buffer size for the computation can be obtained by hipsparseXcsrgemm2_bufferSizeExt().

Note: If $\alpha == 0$, then $C = \beta \cdot D$ will be computed.

Note: If $\beta == 0$, then $C = \alpha \cdot A \cdot B$ will be computed.

Note: $\alpha == beta == 0$ is invalid.

Note: Currently, only HIPSPARSE_MATRIX_TYPE_GENERAL is supported.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

Note: Please note, that for matrix products with more than 4096 non-zero entries per row, additional temporary storage buffer is allocated by the algorithm.

1.12 Preconditioner Functions

This module holds all sparse preconditioners.

The sparse preconditioners describe manipulations on a matrix in sparse format to obtain a sparse preconditioner matrix.

1.12.1 hipsparseXbsrilu02_zeroPivot()

hipsparseStatus_t **hipsparseXbsrilu02_zeroPivot**(*hipsparseHandle_t* handle, *bsrilu02Info_t* info, int *position)

Incomplete LU factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsrilu02_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXbsrilu02_analysis() or hipsparseXbsrilu02() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the BSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: If a zero pivot is found, `position` $= j$ means that either the diagonal block $A_{j,j}$ is missing (structural zero) or the diagonal block $A_{j,j}$ is not invertible (numerical zero).

Note: `hipsparseXbsrilu02_zeroPivot` is a blocking function. It might influence performance negatively.

1.12.2 hipsparseXbsrilu02_numericBoost()

hipsparseStatus_t **hipsparseSbsrilu02_numericBoost**(*hipsparseHandle_t* handle, *bsrilu02Info_t* info, int enable_boost, double *tol, float *boost_val)

hipsparseStatus_t **hipsparseDbsrilu02_numericBoost**(*hipsparseHandle_t* handle, *bsrilu02Info_t* info, int enable_boost, double *tol, double *boost_val)

hipsparseStatus_t **hipsparseCbsrilu02_numericBoost**(*hipsparseHandle_t* handle, *bsrilu02Info_t* info, int enable_boost, double *tol, hipComplex *boost_val)

hipsparseStatus_t **hipsparseZbsrilu02_numericBoost**(*hipsparseHandle_t* handle, *bsrilu02Info_t* info, int enable_boost, double *tol, hipDoubleComplex *boost_val)

Incomplete LU factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsrilu02_numericBoost` enables the user to replace a numerical value in an incomplete LU factorization. `tol` is used to determine whether a numerical value is replaced by `boost_val`, such that $A_{j,j} =$ boost_val if tol $\geq |A_{j,j}|$.

Note: The boost value is enabled by setting `enable_boost` to 1 or disabled by setting `enable_boost` to 0.

Note: `tol` and `boost_val` can be in host or device memory.

1.12.3 hipsparseXbsrilu02_bufferSize()

hipsparseStatus_t **hipsparseSbsrilu02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDbsrilu02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCbsrilu02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZbsrilu02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, int *pBufferSizeInBytes)

Incomplete LU factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsrilu02_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXbsrilu02_analysis() and hipsparseXbsrilu02_solve(). The temporary storage buffer must be allocated by the user.

1.12.4 hipsparseXbsrilu02_analysis()

hipsparseStatus_t **hipsparseSbsrilu02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrilu02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrilu02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrilu02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete LU factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsrilu02_analysis` performs the analysis step for hipsparseXbsrilu02().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.5 hipsparseXbsrilu02()

hipsparseStatus_t **hipsparseSbsrilu02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrSortedValA_valM, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsrilu02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrSortedValA_valM, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsrilu02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrSortedValA_valM, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsrilu02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrSortedValA_valM, const int *bsrSortedRowPtrA, const int *bsrSortedColIndA, int blockDim, *bsrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete LU factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsrilu02` computes the incomplete LU factorization with 0 fill-ins and no pivoting of a sparse $mb \times mb$ BSR matrix A, such that

$$A \approx LU$$

`hipsparseXbsrilu02` requires a user allocated temporary buffer. Its size is returned by hipsparseXbsrilu02_bufferSize(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXbsrilu02_analysis(). `hipsparseXbsrilu02` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be obtained by calling *hipsparseXbsrilu02_zeroPivot()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.6 hipsparseXcsrilu02_zeroPivot()

hipsparseStatus_t **hipsparseXcsrilu02_zeroPivot**(*hipsparseHandle_t* handle, *csrilu02Info_t* info, int *position)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXcsrilu02() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the CSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXcsrilu02_zeroPivot` is a blocking function. It might influence performance negatively.

1.12.7 hipsparseXcsrilu02_numericBoost()

hipsparseStatus_t **hipsparseScsrilu02_numericBoost**(*hipsparseHandle_t* handle, *csrilu02Info_t* info, int enable_boost, double *tol, float *boost_val)

hipsparseStatus_t **hipsparseDcsrilu02_numericBoost**(*hipsparseHandle_t* handle, *csrilu02Info_t* info, int enable_boost, double *tol, double *boost_val)

hipsparseStatus_t **hipsparseCcsrilu02_numericBoost**(*hipsparseHandle_t* handle, *csrilu02Info_t* info, int enable_boost, double *tol, hipComplex *boost_val)

hipsparseStatus_t **hipsparseZcsrilu02_numericBoost**(*hipsparseHandle_t* handle, *csrilu02Info_t* info, int enable_boost, double *tol, hipDoubleComplex *boost_val)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02_numericBoost` enables the user to replace a numerical value in an incomplete LU factorization. `tol` is used to determine whether a numerical value is replaced by `boost_val`, such that $A_{j,j} =$ boost_val if tol $\geq |A_{j,j}|$.

Note: The boost value is enabled by setting `enable_boost` to 1 or disabled by setting `enable_boost` to 0.

Note: `tol` and `boost_val` can be in host or device memory.

1.12.8 hipsparseXcsrilu02_bufferSize()

hipsparseStatus_t **hipsparseScsrilu02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsrilu02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsrilu02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsrilu02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, int *pBufferSizeInBytes)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXcsrilu02_analysis() and hipsparseXcsrilu02_solve(). the temporary storage buffer must be allocated by the user.

1.12.9 hipsparseXcsrilu02_bufferSizeExt()

hipsparseStatus_t **hipsparseScsrilu02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseDcsrilu02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseCcsrilu02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseZcsrilu02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, size_t *pBufferSize)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXcsrilu02_analysis() and hipsparseXcsrilu02_solve(). the temporary storage buffer must be allocated by the user.

1.12.10 hipsparseXcsrilu02_analysis()

hipsparseStatus_t **hipsparseScsrilu02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrilu02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrilu02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrilu02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02_analysis` performs the analysis step for hipsparseXcsrilu02().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.11 hipsparseXcsrilu02()

hipsparseStatus_t **hipsparseScsrilu02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsrilu02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsrilu02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsrilu02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csrilu02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete LU factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsrilu02` computes the incomplete LU factorization with 0 fill-ins and no pivoting of a sparse $m \times m$ CSR matrix A, such that

$$A \approx LU$$

`hipsparseXcsrilu02` requires a user allocated temporary buffer. Its size is returned by hipsparseXcsrilu02_bufferSize() or hipsparseXcsrilu02_bufferSizeExt(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXcsrilu02_analysis(). `hipsparseXcsrilu02` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be obtained by calling *hipsparseXcsrilu02_zeroPivot()*.

Note: The sparse CSR matrix has to be sorted. This can be achieved by calling *hipsparseXcsrsort()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.12 hipsparseXbsric02_zeroPivot()

hipsparseStatus_t **hipsparseXbsric02_zeroPivot**(*hipsparseHandle_t* handle, *bsric02Info_t* info, int *position)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsric02_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXbsric02_analysis() or hipsparseXbsric02() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the BSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: If a zero pivot is found, `position=j` means that either the diagonal block `A(j,j)` is missing (structural zero) or the diagonal block `A(j,j)` is not positive definite (numerical zero).

Note: `hipsparseXbsric02_zeroPivot` is a blocking function. It might influence performance negatively.

1.12.13 hipsparseXbsric02_bufferSize()

hipsparseStatus_t **hipsparseSbsric02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDbsric02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCbsric02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZbsric02_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, int *pBufferSizeInBytes)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsric02_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXbsric02_analysis() and hipsparseXbsric02(). The temporary storage buffer must be allocated by the user.

1.12.14 hipsparseXbsric02_analysis()

hipsparseStatus_t **hipsparseSbsric02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsric02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsric02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsric02_analysis**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsric02_analysis` performs the analysis step for hipsparseXbsric02().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.15 hipsparseXbsric02()

hipsparseStatus_t **hipsparseSbsric02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDbsric02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCbsric02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZbsric02**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nnzb, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, *bsric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using BSR storage format.

`hipsparseXbsric02` computes the incomplete Cholesky factorization with 0 fill-ins and no pivoting of a sparse $mb \times mb$ BSR matrix A, such that

$$A \approx LL^T$$

`hipsparseXbsric02` requires a user allocated temporary buffer. Its size is returned by hipsparseXbsric02_bufferSize(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXbsric02_analysis(). `hipsparseXbsric02` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be obtained by calling *hipsparseXbsric02_zeroPivot()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.16 hipsparseXcsric02_zeroPivot()

hipsparseStatus_t **hipsparseXcsric02_zeroPivot**(*hipsparseHandle_t* handle, *csric02Info_t* info, int *position)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsric02_zeroPivot` returns HIPSPARSE_STATUS_ZERO_PIVOT, if either a structural or numerical zero has been found during hipsparseXcsric02_analysis() or hipsparseXcsric02() computation. The first zero pivot j at $A_{j,j}$ is stored in `position`, using same index base as the CSR matrix.

`position` can be in host or device memory. If no zero pivot has been found, `position` is set to -1 and HIPSPARSE_STATUS_SUCCESS is returned instead.

Note: `hipsparseXcsric02_zeroPivot` is a blocking function. It might influence performance negatively.

1.12.17 hipsparseXcsric02_bufferSize()

hipsparseStatus_t **hipsparseScsric02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsric02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsric02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, int *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsric02_bufferSize**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, int *pBufferSizeInBytes)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsric02_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXcsric02_analysis() and hipsparseXcsric02().

1.12.18 hipsparseXcsric02_bufferSizeExt()

hipsparseStatus_t **hipsparseScsric02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseDcsric02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseCcsric02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, size_t *pBufferSize)

hipsparseStatus_t **hipsparseZcsric02_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, size_t *pBufferSize)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsric02_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXcsric02_analysis() and hipsparseXcsric02().

1.12.19 hipsparseXcsric02_analysis()

hipsparseStatus_t **hipsparseScsric02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsric02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsric02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsric02_analysis**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsric02_analysis` performs the analysis step for hipsparseXcsric02().

Note: If the matrix sparsity pattern changes, the gathered information will become invalid.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.20 hipsparseXcsric02()

hipsparseStatus_t **hipsparseScsric02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, float *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseDcsric02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, double *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseCcsric02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

hipsparseStatus_t **hipsparseZcsric02**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrSortedValA_valM, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *csric02Info_t* info, *hipsparseSolvePolicy_t* policy, void *pBuffer)

Incomplete Cholesky factorization with 0 fill-ins and no pivoting using CSR storage format.

`hipsparseXcsric02` computes the incomplete Cholesky factorization with 0 fill-ins and no pivoting of a sparse $m \times m$ CSR matrix A, such that

$$A \approx LL^T$$

`hipsparseXcsric02` requires a user allocated temporary buffer. Its size is returned by hipsparseXcsric02_bufferSize() or hipsparseXcsric02_bufferSizeExt(). Furthermore, analysis meta data is required. It can be obtained by hipsparseXcsric02_analysis(). `hipsparseXcsric02` reports the first zero pivot (either numerical or structural zero). The zero pivot status can be obtained by calling *hipsparseXcsric02_zeroPivot()*.

Note: The sparse CSR matrix has to be sorted. This can be achieved by calling *hipsparseXcsrsort()*.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.21 hipsparseXgtsv2_bufferSizeExt()

hipsparseStatus_t **hipsparseSgtsv2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const float *dl, const float *d, const float *du, const float *B, int ldb, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDgtsv2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const double *dl, const double *d, const double *du, const double *B, int db, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCgtsv2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *dl, const hipComplex *d, const hipComplex *du, const hipComplex *B, int ldb, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZgtsv2_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, const hipDoubleComplex *B, int ldb, size_t *pBufferSizeInBytes)

Tridiagonal solver with pivoting.

`hipsparseXgtsv2_bufferSize` returns the size of the temporary storage buffer that is required by hipsparseXgtsv2(). The temporary storage buffer must be allocated by the user.

1.12.22 hipsparseXgtsv2()

hipsparseStatus_t **hipsparseSgtsv2**(*hipsparseHandle_t* handle, int m, int n, const float *dl, const float *d, const float *du, float *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseDgtsv2**(*hipsparseHandle_t* handle, int m, int n, const double *dl, const double *d, const double *du, double *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseCgtsv2**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *dl, const hipComplex *d, const hipComplex *du, hipComplex *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseZgtsv2**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, hipDoubleComplex *B, int ldb, void *pBuffer)

Tridiagonal solver with pivoting.

`hipsparseXgtsv2` solves a tridiagonal system for multiple right hand sides using pivoting.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.23 hipsparseXgtsv2_nopivot_bufferSizeExt()

hipsparseStatus_t **hipsparseSgtsv2_nopivot_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const float *dl, const float *d, const float *du, const float *B, int ldb, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDgtsv2_nopivot_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const double *dl, const double *d, const double *du, const double *B, int db, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCgtsv2_nopivot_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *dl, const hipComplex *d, const hipComplex *du, const hipComplex *B, int ldb, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZgtsv2_nopivot_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, const hipDoubleComplex *B, int ldb, size_t *pBufferSizeInBytes)

Tridiagonal solver (no pivoting)

`hipsparseXgtsv2_nopivot_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXgtsv2_nopivot(). The temporary storage buffer must be allocated by the user.

1.12.24 hipsparseXgtsv2_nopivot()

hipsparseStatus_t **hipsparseSgtsv2_nopivot**(*hipsparseHandle_t* handle, int m, int n, const float *dl, const float *d, const float *du, float *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseDgtsv2_nopivot**(*hipsparseHandle_t* handle, int m, int n, const double *dl, const double *d, const double *du, double *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseCgtsv2_nopivot**(*hipsparseHandle_t* handle, int m, int n, const hipComplex *dl, const hipComplex *d, const hipComplex *du, hipComplex *B, int ldb, void *pBuffer)

hipsparseStatus_t **hipsparseZgtsv2_nopivot**(*hipsparseHandle_t* handle, int m, int n, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, hipDoubleComplex *B, int ldb, void *pBuffer)

Tridiagonal solver (no pivoting)

`hipsparseXgtsv2_nopivot` solves a tridiagonal linear system for multiple right-hand sides

---

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.25 hipsparseXgtsv2StridedBatch_bufferSizeExt()

hipsparseStatus_t **hipsparseSgtsv2StridedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int m, const float *dl, const float *d, const float *du, const float *x, int batchCount, int batchStride, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDgtsv2StridedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int m, const double *dl, const double *d, const double *du, const double *x, int batchCount, int batchStride, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCgtsv2StridedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int m, const hipComplex *dl, const hipComplex *d, const hipComplex *du, const hipComplex *x, int batchCount, int batchStride, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZgtsv2StridedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int m, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, const hipDoubleComplex *x, int batchCount, int batchStride, size_t *pBufferSizeInBytes)

Strided Batch tridiagonal solver (no pivoting)

`hipsparseXgtsv2StridedBatch_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXgtsv2StridedBatch(). The temporary storage buffer must be allocated by the user.

1.12.26 hipsparseXgtsv2StridedBatch()

hipsparseStatus_t **hipsparseSgtsv2StridedBatch**(*hipsparseHandle_t* handle, int m, const float *dl, const float *d, const float *du, float *x, int batchCount, int batchStride, void *pBuffer)

hipsparseStatus_t **hipsparseDgtsv2StridedBatch**(*hipsparseHandle_t* handle, int m, const double *dl, const double *d, const double *du, double *x, int batchCount, int batchStride, void *pBuffer)

hipsparseStatus_t **hipsparseCgtsv2StridedBatch**(*hipsparseHandle_t* handle, int m, const hipComplex *dl, const hipComplex *d, const hipComplex *du, hipComplex *x, int batchCount, int batchStride, void *pBuffer)

hipsparseStatus_t **hipsparseZgtsv2StridedBatch**(*hipsparseHandle_t* handle, int m, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, hipDoubleComplex *x, int batchCount, int batchStride, void *pBuffer)

Strided Batch tridiagonal solver (no pivoting)

`hipsparseXgtsv2StridedBatch` solves a batched tridiagonal linear system

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.27 hipsparseXgtsvInterleavedBatch_bufferSizeExt()

hipsparseStatus_t **hipsparseSgtsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const float *dl, const float *d, const float *du, const float *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDgtsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const double *dl, const double *d, const double *du, const double *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCgtsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const hipComplex *dl, const hipComplex *d, const hipComplex *du, const hipComplex *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZgtsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, const hipDoubleComplex *x, int batchCount, size_t *pBufferSizeInBytes)

Interleaved Batch tridiagonal solver.

`hipsparseXgtsvInterleavedBatch_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXgtsvInterleavedBatch(). The temporary storage buffer must be allocated by the user.

1.12.28 hipsparseXgtsvInterleavedBatch()

hipsparseStatus_t **hipsparseSgtsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, float *dl, float *d, float *du, float *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseDgtsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, double *dl, double *d, double *du, double *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseCgtsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, hipComplex *dl, hipComplex *d, hipComplex *du, hipComplex *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseZgtsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, hipDoubleComplex *dl, hipDoubleComplex *d, hipDoubleComplex *du, hipDoubleComplex *x, int batchCount, void *pBuffer)

Interleaved Batch tridiagonal solver.

`hipsparseXgtsvInterleavedBatch` solves a batched tridiagonal linear system

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.12.29 hipsparseXgpsvInterleavedBatch_bufferSizeExt()

hipsparseStatus_t **hipsparseSgpsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const float *ds, const float *dl, const float *d, const float *du, const float *dw, const float *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDgpsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const double *ds, const double *dl, const double *d, const double *du, const double *dw, const double *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCgpsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const hipComplex *ds, const hipComplex *dl, const hipComplex *d, const hipComplex *du, const hipComplex *dw, const hipComplex *x, int batchCount, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZgpsvInterleavedBatch_bufferSizeExt**(*hipsparseHandle_t* handle, int algo, int m, const hipDoubleComplex *ds, const hipDoubleComplex *dl, const hipDoubleComplex *d, const hipDoubleComplex *du, const hipDoubleComplex *dw, const hipDoubleComplex *x, int batchCount, size_t *pBufferSizeInBytes)

Interleaved Batch pentadiagonal solver.

`hipsparseXgpsvInterleavedBatch_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXgpsvInterleavedBatch(). The temporary storage buffer must be allocated by the user.

1.12.30 hipsparseXgpsvInterleavedBatch()

hipsparseStatus_t **hipsparseSgpsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, float *ds, float *dl, float *d, float *du, float *dw, float *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseDgpsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, double *ds, double *dl, double *d, double *du, double *dw, double *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseCgpsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, hipComplex *ds, hipComplex *dl, hipComplex *d, hipComplex *du, hipComplex *dw, hipComplex *x, int batchCount, void *pBuffer)

hipsparseStatus_t **hipsparseZgpsvInterleavedBatch**(*hipsparseHandle_t* handle, int algo, int m, hipDoubleComplex *ds, hipDoubleComplex *dl, hipDoubleComplex *d, hipDoubleComplex *du, hipDoubleComplex *dw, hipDoubleComplex *x, int batchCount, void *pBuffer)

Interleaved Batch pentadiagonal solver.

`hipsparseXgpsvInterleavedBatch` solves a batched pentadiagonal linear system

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13 Sparse Conversion Functions

This module holds all sparse conversion routines.

The sparse conversion routines describe operations on a matrix in sparse format to obtain a matrix in a different sparse format.

1.13.1 hipsparseXnnz()

hipsparseStatus_t **hipsparseSnnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const float *A, int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)

hipsparseStatus_t **hipsparseDnnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const double *A, int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)

hipsparseStatus_t **hipsparseCnnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const hipComplex *A, int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)

hipsparseStatus_t **hipsparseZnnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *A, int lda, int *nnzPerRowColumn, int *nnzTotalDevHostPtr)

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

The routine does support asynchronous execution if the pointer mode is set to device.

1.13.2 hipsparseXdense2csr()

hipsparseStatus_t **hipsparseSdense2csr**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const float *A, int ld, const int *nnz_per_rows, float *csr_val, int *csr_row_ptr, int *csr_col_ind)

hipsparseStatus_t **hipsparseDdense2csr**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const double *A, int ld, const int *nnz_per_rows, double *csr_val, int *csr_row_ptr, int *csr_col_ind)

hipsparseStatus_t **hipsparseCdense2csr**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipComplex *A, int ld, const int *nnz_per_rows, hipComplex *csr_val, int *csr_row_ptr, int *csr_col_ind)

hipsparseStatus_t **hipsparseZdense2csr**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipDoubleComplex *A, int ld, const int *nnz_per_rows, hipDoubleComplex *csr_val, int *csr_row_ptr, int *csr_col_ind)

This function converts the matrix A in dense format into a sparse matrix in CSR format. All the parameters are assumed to have been pre-allocated by the user and the arrays are filled in based on nnz_per_row, which can be pre-computed with hipsparseXnnz(). It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.3 hipsparseXpruneDense2csr_bufferSize()

hipsparseStatus_t **hipsparseSpruneDense2csr_bufferSize**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, const float *threshold, const *hipsparseMatDescr_t* descr, const float *csrVal, const int *csrRowPtr, const int *csrColInd, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneDense2csr_bufferSize**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, const double *threshold, const *hipsparseMatDescr_t* descr, const double *csrVal, const int *csrRowPtr, const int *csrColInd, size_t *bufferSize)

This function computes the the size of the user allocated temporary storage buffer used when converting and pruning a dense matrix to a CSR matrix.

`hipsparseXpruneDense2csr_bufferSizeExt` returns the size of the temporary storage buffer that is required by hipsparseXpruneDense2csrNnz() and hipsparseXpruneDense2csr(). The temporary storage buffer must be allocated by the user.

1.13.4 hipsparseXpruneDense2csrNnz()

hipsparseStatus_t **hipsparseSpruneDense2csrNnz**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, const float *threshold, const *hipsparseMatDescr_t* descr, int *csrRowPtr, int *nnzTotalDevHostPtr, void *buffer)

hipsparseStatus_t **hipsparseDpruneDense2csrNnz**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, const double *threshold, const *hipsparseMatDescr_t* descr, int *csrRowPtr, int *nnzTotalDevHostPtr, void *buffer)

This function computes the number of nonzero elements per row and the total number of nonzero elements in a dense matrix once elements less than the threshold are pruned from the matrix.

The routine does support asynchronous execution if the pointer mode is set to device.

1.13.5 hipsparseXpruneDense2csr()

hipsparseStatus_t **hipsparseSpruneDense2csr**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, const float *threshold, const *hipsparseMatDescr_t* descr, float *csrVal, const int *csrRowPtr, int *csrColInd, void *buffer)

hipsparseStatus_t **hipsparseDpruneDense2csr**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, const double *threshold, const *hipsparseMatDescr_t* descr, double *csrVal, const int *csrRowPtr, int *csrColInd, void *buffer)

This function converts the matrix A in dense format into a sparse matrix in CSR format while pruning values that are less than the threshold. All the parameters are assumed to have been pre-allocated by the user.

The user first allocates `csrRowPtr` to have `m+1` elements and then calls hipsparseXpruneDense2csrNnz() which fills in the `csrRowPtr` array and stores the number of elements that are larger than the pruning threshold in `nnzTotalDevHostPtr`. The user then allocates `csrColInd` and `csrVal` to have size `nnzTotalDevHostPtr` and completes the conversion by calling hipsparseXpruneDense2csr(). A temporary storage buffer is used by both hipsparseXpruneDense2csrNnz() and hipsparseXpruneDense2csr() and must be allocated by the user and whose size is determined by hipsparseXpruneDense2csr_bufferSizeExt(). The routine hipsparseXpruneDense2csr() is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.6 hipsparseXpruneDense2csrByPercentage_bufferSize()

hipsparseStatus_t **hipsparseSpruneDense2csrByPercentage_bufferSize**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, float percentage, const *hipsparseMatDescr_t* descr, const float *csrVal, const int *csrRowPtr, const int *csrColInd, *pruneInfo_t* info, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneDense2csrByPercentage_bufferSize**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, double percentage, const *hipsparseMatDescr_t* descr, const double *csrVal, const int *csrRowPtr, const int *csrColInd, *pruneInfo_t* info, size_t *bufferSize)

This function computes the size of the user allocated temporary storage buffer used when converting and pruning by percentage a dense matrix to a CSR matrix.

When converting and pruning a dense matrix A to a CSR matrix by percentage the following steps are performed. First the user calls `hipsparseXpruneDense2csrByPercentage_bufferSize` which determines the size of the temporary storage buffer. Once determined, this buffer must be allocated by the user. Next the user allocates the csr_row_ptr array to have `m+1` elements and calls `hipsparseXpruneDense2csrNnzByPercentage`. Finally

the user finishes the conversion by allocating the csr_col_ind and csr_val arrays (whos size is determined by the value at nnz_total_dev_host_ptr) and calling `hipsparseXpruneDense2csrByPercentage`.

The pruning by percentage works by first sorting the absolute values of the dense matrix `A`. We then determine a position in this sorted array by

$$pos = ceil(m * n * (percentage/100)) - 1 pos = min(pos, m * n - 1) pos = max(pos, 0) threshold = sorted_A[pos]$$

Once we have this threshold we prune values in the dense matrix `A` as in `hipsparseXpruneDense2csr`. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.7 hipsparseXpruneDense2csrByPercentage_bufferSizeExt()

hipsparseStatus_t **hipsparseSpruneDense2csrByPercentage_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, float percentage, const *hipsparseMatDescr_t* descr, const float *csrVal, const int *csrRowPtr, const int *csrColInd, *pruneInfo_t* info, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneDense2csrByPercentage_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, double percentage, const *hipsparseMatDescr_t* descr, const double *csrVal, const int *csrRowPtr, const int *csrColInd, *pruneInfo_t* info, size_t *bufferSize)

This function computes the size of the user allocated temporary storage buffer used when converting and pruning by percentage a dense matrix to a CSR matrix.

When converting and pruning a dense matrix A to a CSR matrix by percentage the following steps are performed. First the user calls `hipsparseXpruneDense2csrByPercentage_bufferSizeExt` which determines the size of the temporary storage buffer. Once determined, this buffer must be allocated by the user. Next the user allocates the csr_row_ptr array to have `m+1` elements and calls `hipsparseXpruneDense2csrNnzByPercentage`. Finally the user finishes the conversion by allocating the csr_col_ind and csr_val arrays (whos size is determined by the value at nnz_total_dev_host_ptr) and calling `hipsparseXpruneDense2csrByPercentage`.

The pruning by percentage works by first sorting the absolute values of the dense matrix `A`. We then determine a position in this sorted array by

$$pos = ceil(m * n * (percentage/100)) - 1 pos = min(pos, m * n - 1) pos = max(pos, 0) threshold = sorted_A[pos]$$

Once we have this threshold we prune values in the dense matrix `A` as in `hipsparseXpruneDense2csr`. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.8 hipsparseXpruneDense2csrNnzByPercentage()

hipsparseStatus_t **hipsparseSpruneDense2csrNnzByPercentage**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, float percentage, const *hipsparseMatDescr_t* descr, int *csrRowPtr, int *nnzTotalDevHostPtr, *pruneInfo_t* info, void *buffer)

hipsparseStatus_t **hipsparseDpruneDense2csrNnzByPercentage**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, double percentage, const *hipsparseMatDescr_t* descr, int *csrRowPtr, int *nnzTotalDevHostPtr, *pruneInfo_t* info, void *buffer)

This function computes the number of nonzero elements per row and the total number of nonzero elements in a dense matrix when converting and pruning by percentage a dense matrix to a CSR matrix.

When converting and pruning a dense matrix A to a CSR matrix by percentage the following steps are performed. First the user calls `hipsparseXpruneDense2csrByPercentage_bufferSize` which determines the size of the temporary storage buffer. Once determined, this buffer must be allocated by the user. Next the user allocates the csr_row_ptr array to have `m+1` elements and calls `hipsparseXpruneDense2csrNnzByPercentage`. Finally the user finishes the conversion by allocating the csr_col_ind and csr_val arrays (whos size is determined by the value at nnz_total_dev_host_ptr) and calling `hipsparseXpruneDense2csrByPercentage`.

The pruning by percentage works by first sorting the absolute values of the dense matrix `A`. We then determine a position in this sorted array by

$$pos = ceil(m * n * (percentage/100)) - 1 pos = min(pos, m * n - 1) pos = max(pos, 0) threshold = sorted_A[pos]$$

Once we have this threshold we prune values in the dense matrix `A` as in `hipsparseXpruneDense2csr`. The routine does support asynchronous execution if the pointer mode is set to device.

1.13.9 hipsparseXpruneDense2csrByPercentage()

hipsparseStatus_t **hipsparseSpruneDense2csrByPercentage**(*hipsparseHandle_t* handle, int m, int n, const float *A, int lda, float percentage, const *hipsparseMatDescr_t* descr, float *csrVal, const int *csrRowPtr, int *csrColInd, *pruneInfo_t* info, void *buffer)

hipsparseStatus_t **hipsparseDpruneDense2csrByPercentage**(*hipsparseHandle_t* handle, int m, int n, const double *A, int lda, double percentage, const *hipsparseMatDescr_t* descr, double *csrVal, const int *csrRowPtr, int *csrColInd, *pruneInfo_t* info, void *buffer)

This function computes the number of nonzero elements per row and the total number of nonzero elements in a dense matrix when converting and pruning by percentage a dense matrix to a CSR matrix.

When converting and pruning a dense matrix A to a CSR matrix by percentage the following steps are performed. First the user calls `hipsparseXpruneDense2csrByPercentage_bufferSize` which determines the size of the temporary storage buffer. Once determined, this buffer must be allocated by the user. Next the user allocates the csr_row_ptr array to have `m+1` elements and calls `hipsparseXpruneDense2csrNnzByPercentage`. Finally the user finishes the conversion by allocating the csr_col_ind and csr_val arrays (whos size is determined by the value at nnz_total_dev_host_ptr) and calling `hipsparseXpruneDense2csrByPercentage`.

The pruning by percentage works by first sorting the absolute values of the dense matrix `A`. We then determine a position in this sorted array by

$$pos = ceil(m * n * (percentage/100)) - 1 pos = min(pos, m * n - 1) pos = max(pos, 0) threshold = sorted_A[pos]$$

Once we have this threshold we prune values in the dense matrix `A` as in `hipsparseXpruneDense2csr`. The routine does support asynchronous execution if the pointer mode is set to device.

1.13.10 hipsparseXdense2csc()

hipsparseStatus_t **hipsparseSdense2csc**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const float *A, int ld, const int *nnz_per_columns, float *csc_val, int *csc_row_ind, int *csc_col_ptr)

hipsparseStatus_t **hipsparseDdense2csc**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const double *A, int ld, const int *nnz_per_columns, double *csc_val, int *csc_row_ind, int *csc_col_ptr)

hipsparseStatus_t **hipsparseCdense2csc**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipComplex *A, int ld, const int *nnz_per_columns, hipComplex *csc_val, int *csc_row_ind, int *csc_col_ptr)

hipsparseStatus_t **hipsparseZdense2csc**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipDoubleComplex *A, int ld, const int *nnz_per_columns, hipDoubleComplex *csc_val, int *csc_row_ind, int *csc_col_ptr)

This function converts the matrix A in dense format into a sparse matrix in CSC format. All the parameters are assumed to have been pre-allocated by the user and the arrays are filled in based on nnz_per_columns, which can be pre-computed with hipsparseXnnz(). It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.11 hipsparseXcsr2dense()

hipsparseStatus_t **hipsparseScsr2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const float *csr_val, const int *csr_row_ptr, const int *csr_col_ind, float *A, int ld)

hipsparseStatus_t **hipsparseDcsr2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const double *csr_val, const int *csr_row_ptr, const int *csr_col_ind, double *A, int ld)

hipsparseStatus_t **hipsparseCcsr2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, hipComplex *A, int ld)

hipsparseStatus_t **hipsparseZcsr2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipDoubleComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, hipDoubleComplex *A, int ld)

This function converts the sparse matrix in CSR format into a dense matrix. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.12 hipsparseXcsc2dense()

hipsparseStatus_t **hipsparseScsc2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const float *csc_val, const int *csc_row_ind, const int *csc_col_ptr, float *A, int ld)

hipsparseStatus_t **hipsparseDcsc2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const double *csc_val, const int *csc_row_ind, const int *csc_col_ptr, double *A, int ld)

hipsparseStatus_t **hipsparseCcsc2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipComplex *csc_val, const int *csc_row_ind, const int *csc_col_ptr, hipComplex *A, int ld)

hipsparseStatus_t **hipsparseZcsc2dense**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descr, const hipDoubleComplex *csc_val, const int *csc_row_ind, const int *csc_col_ptr, hipDoubleComplex *A, int ld)

This function converts the sparse matrix in CSC format into a dense matrix. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.13 hipsparseXcsr2bsrNnz()

hipsparseStatus_t **hipsparseXcsr2bsrNnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const int *csrRowPtrA, const int *csrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, int *bsrRowPtrC, int *bsrNnzb)

This function computes the number of nonzero block columns per row and the total number of nonzero blocks in a sparse BSR matrix given a sparse CSR matrix as input.

The routine does support asynchronous execution if the pointer mode is set to device.

1.13.14 hipsparseXcsr2bsr()

hipsparseStatus_t **hipsparseScsr2bsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, float *bsrValC, int *bsrRowPtrC, int *bsrColIndC)

hipsparseStatus_t **hipsparseDcsr2bsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, double *bsrValC, int *bsrRowPtrC, int *bsrColIndC)

hipsparseStatus_t **hipsparseCcsr2bsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const hipComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, hipComplex *bsrValC, int *bsrRowPtrC, int *bsrColIndC)

hipsparseStatus_t **hipsparseZcsr2bsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int m, int n, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *bsrValC, int *bsrRowPtrC, int *bsrColIndC)

Convert a sparse CSR matrix into a sparse BSR matrix.

`hipsparseXcsr2bsr` converts a CSR matrix into a BSR matrix. It is assumed, that `bsr_val`, `bsr_col_ind` and `bsr_row_ptr` are allocated. Allocation size for `bsr_row_ptr` is computed as `mb+1` where `mb` is the number of block rows in the BSR matrix. Allocation size for `bsr_val` and `bsr_col_ind` is computed using `csr2bsr_nnz()` which also fills in `bsr_row_ptr`.

`hipsparseXcsr2bsr` requires extra temporary storage that is allocated internally if `block_dim>16`

1.13.15 hipsparseXnnz_compress()

hipsparseStatus_t **hipsparseSnnz_compress**(*hipsparseHandle_t* handle, int m, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, int *nnzPerRow, int *nnzC, float tol)

hipsparseStatus_t **hipsparseDnnz_compress**(*hipsparseHandle_t* handle, int m, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, int *nnzPerRow, int *nnzC, double tol)

hipsparseStatus_t **hipsparseCnnz_compress**(*hipsparseHandle_t* handle, int m, const *hipsparseMatDescr_t* descrA, const hipComplex *csrValA, const int *csrRowPtrA, int *nnzPerRow, int *nnzC, hipComplex tol)

hipsparseStatus_t **hipsparseZnnz_compress**(*hipsparseHandle_t* handle, int m, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, int *nnzPerRow, int *nnzC, hipDoubleComplex tol)

Given a sparse CSR matrix and a non-negative tolerance, this function computes how many entries would be left in each row of the matrix if elements less than the tolerance were removed. It also computes the total number of remaining elements in the matrix.

1.13.16 hipsparseXcsr2coo()

hipsparseStatus_t **hipsparseXcsr2coo**(*hipsparseHandle_t* handle, const int *csrRowPtr, int nnz, int m, int *cooRowInd, *hipsparseIndexBase_t* idxBase)

Convert a sparse CSR matrix into a sparse COO matrix.

`hipsparseXcsr2coo` converts the CSR array containing the row offsets, that point to the start of every row, into a COO array of row indices.

Note: It can also be used to convert a CSC array containing the column offsets into a COO array of column indices.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.17 hipsparseXcsr2csc()

hipsparseStatus_t **hipsparseScsr2csc**(*hipsparseHandle_t* handle, int m, int n, int nnz, const float *csrSortedVal, const int *csrSortedRowPtr, const int *csrSortedColInd, float *cscSortedVal, int *cscSortedRowInd, int *cscSortedColPtr, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseDcsr2csc**(*hipsparseHandle_t* handle, int m, int n, int nnz, const double *csrSortedVal, const int *csrSortedRowPtr, const int *csrSortedColInd, double *cscSortedVal, int *cscSortedRowInd, int *cscSortedColPtr, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseCcsr2csc**(*hipsparseHandle_t* handle, int m, int n, int nnz, const hipComplex *csrSortedVal, const int *csrSortedRowPtr, const int *csrSortedColInd, hipComplex *cscSortedVal, int *cscSortedRowInd, int *cscSortedColPtr, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase)

hipsparseStatus_t **hipsparseZcsr2csc**(*hipsparseHandle_t* handle, int m, int n, int nnz, const hipDoubleComplex *csrSortedVal, const int *csrSortedRowPtr, const int *csrSortedColInd, hipDoubleComplex *cscSortedVal, int *cscSortedRowInd, int *cscSortedColPtr, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase)

Convert a sparse CSR matrix into a sparse CSC matrix.

`hipsparseXcsr2csc` converts a CSR matrix into a CSC matrix. `hipsparseXcsr2csc` can also be used to convert a CSC matrix into a CSR matrix. `copy_values` decides whether `csc_val` is being filled during conversion (HIPSPARSE_ACTION_NUMERIC) or not (HIPSPARSE_ACTION_SYMBOLIC).

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.18 hipsparseXcsr2cscEx2_bufferSize()

hipsparseStatus_t **hipsparseCsr2cscEx2_bufferSize**(*hipsparseHandle_t* handle, int m, int n, int nnz, const void *csrVal, const int *csrRowPtr, const int *csrColInd, void *cscVal, int *cscColPtr, int *cscRowInd, hipDataType valType, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase, *hipsparseCsr2CscAlg_t* alg, size_t *bufferSize)

This function computes the size of the user allocated temporary storage buffer used when converting a sparse CSR matrix into a sparse CSC matrix.

`hipsparseXcsr2cscEx2_bufferSize` calculates the required user allocated temporary buffer needed by `hipsparseXcsr2cscEx2` to convert a CSR matrix into a CSC matrix. `hipsparseXcsr2cscEx2` can also be used to convert a CSC matrix into a CSR matrix. `copy_values` decides whether `csc_val` is being filled during conversion (HIPSPARSE_ACTION_NUMERIC) or not (HIPSPARSE_ACTION_SYMBOLIC).

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.19 hipsparseXcsr2cscEx2()

hipsparseStatus_t **hipsparseCsr2cscEx2**(*hipsparseHandle_t* handle, int m, int n, int nnz, const void *csrVal, const int *csrRowPtr, const int *csrColInd, void *cscVal, int *cscColPtr, int *cscRowInd, hipDataType valType, *hipsparseAction_t* copyValues, *hipsparseIndexBase_t* idxBase, *hipsparseCsr2CscAlg_t* alg, void *buffer)

Convert a sparse CSR matrix into a sparse CSC matrix.

`hipsparseXcsr2cscEx2` converts a CSR matrix into a CSC matrix. `hipsparseXcsr2cscEx2` can also be used to convert a CSC matrix into a CSR matrix. `copy_values` decides whether `csc_val` is being filled during conversion (HIPSPARSE_ACTION_NUMERIC) or not (HIPSPARSE_ACTION_SYMBOLIC).

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.20 hipsparseXcsr2hyb()

hipsparseStatus_t **hipsparseScsr2hyb**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const float *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *hipsparseHybMat_t* hybA, int userEllWidth, *hipsparseHybPartition_t* partitionType)

hipsparseStatus_t **hipsparseDcsr2hyb**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const double *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *hipsparseHybMat_t* hybA, int userEllWidth, *hipsparseHybPartition_t* partitionType)

hipsparseStatus_t **hipsparseCcsr2hyb**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const hipComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *hipsparseHybMat_t* hybA, int userEllWidth, *hipsparseHybPartition_t* partitionType)

hipsparseStatus_t **hipsparseZcsr2hyb**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrSortedValA, const int *csrSortedRowPtrA, const int *csrSortedColIndA, *hipsparseHybMat_t* hybA, int userEllWidth, *hipsparseHybPartition_t* partitionType)

Convert a sparse CSR matrix into a sparse HYB matrix.

`hipsparseXcsr2hyb` converts a CSR matrix into a HYB matrix. It is assumed that `hyb` has been initialized with *hipsparseCreateHybMat()*.

Note: This function requires a significant amount of storage for the HYB matrix, depending on the matrix structure.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.21 hipsparseXgebsr2gebsc_bufferSize()

hipsparseStatus_t **hipsparseSgebsr2gebsc_bufferSize**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const float *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseDgebsr2gebsc_bufferSize**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const double *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseCgebsr2gebsc_bufferSize**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const hipComplex *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseZgebsr2gebsc_bufferSize**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const hipDoubleComplex *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

Convert a sparse GEneral BSR matrix into a sparse GEneral BSC matrix.

`hipsparseXgebsr2gebsc_bufferSize` returns the size of the temporary storage buffer required by hipsparseXgebsr2gebsc(). The temporary storage buffer must be allocated by the user.

1.13.22 hipsparseXgebsr2gebsc()

hipsparseStatus_t **hipsparseSgebsr2gebsc**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const float *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, float *bsc_val, int *bsc_row_ind, int *bsc_col_ptr, *hipsparseAction_t* copy_values, *hipsparseIndexBase_t* idx_base, void *temp_buffer)

hipsparseStatus_t **hipsparseDgebsr2gebsc**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const double *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, double *bsc_val, int *bsc_row_ind, int *bsc_col_ptr, *hipsparseAction_t* copy_values, *hipsparseIndexBase_t* idx_base, void *temp_buffer)

hipsparseStatus_t **hipsparseCgebsr2gebsc**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const hipComplex *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, hipComplex *bsc_val, int *bsc_row_ind, int *bsc_col_ptr, *hipsparseAction_t* copy_values, *hipsparseIndexBase_t* idx_base, void *temp_buffer)

hipsparseStatus_t **hipsparseZgebsr2gebsc**(*hipsparseHandle_t* handle, int mb, int nb, int nnzb, const hipDoubleComplex *bsr_val, const int *bsr_row_ptr, const int *bsr_col_ind, int row_block_dim, int col_block_dim, hipDoubleComplex *bsc_val, int *bsc_row_ind, int *bsc_col_ptr, *hipsparseAction_t* copy_values, *hipsparseIndexBase_t* idx_base, void *temp_buffer)

Convert a sparse GEneral BSR matrix into a sparse GEneral BSC matrix.

`hipsparseXgebsr2gebsc` converts a GEneral BSR matrix into a GEneral BSC matrix. `hipsparseXgebsr2gebsc` can also be used to convert a GEneral BSC matrix into a GEneral BSR matrix. `copy_values` decides whether `bsc_val` is being filled during conversion (HIPSPARSE_ACTION_NUMERIC) or not (HIPSPARSE_ACTION_SYMBOLIC).

`hipsparseXgebsr2gebsc` requires extra temporary storage buffer that has to be allocated by the user. Storage buffer size can be determined by hipsparseXgebsr2gebsc_bufferSize().

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.23 hipsparseXcsr2gebsr_bufferSize()

hipsparseStatus_t **hipsparseScsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const float *csr_val, const int *csr_row_ptr, const int *csr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseDcsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const double *csr_val, const int *csr_row_ptr, const int *csr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseCcsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const hipComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

hipsparseStatus_t **hipsparseZcsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const hipDoubleComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, int row_block_dim, int col_block_dim, size_t *p_buffer_size)

`hipsparseXcsr2gebsr_bufferSize` returns the size of the temporary buffer that is required by `hipsparseXcsr2gebcsrNnz` and `hipsparseXcsr2gebcsr`. The temporary storage buffer must be allocated by the user.

This function computes the number of nonzero block columns per row and the total number of nonzero blocks in a sparse GEneral BSR matrix given a sparse CSR matrix as input.

The routine does support asynchronous execution if the pointer mode is set to device.

1.13.24 hipsparseXcsr2gebsrNnz()

hipsparseStatus_t **hipsparseXcsr2gebsrNnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const int *csr_row_ptr, const int *csr_col_ind, const *hipsparseMatDescr_t* bsr_descr, int *bsr_row_ptr, int row_block_dim, int col_block_dim, int *bsr_nnz_devhost, void *p_buffer)

This function computes the number of nonzero block columns per row and the total number of nonzero blocks in a sparse GEneral BSR matrix given a sparse CSR matrix as input.

1.13.25 hipsparseXcsr2gebsr()

hipsparseStatus_t **hipsparseScsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const float *csr_val, const int *csr_row_ptr, const int *csr_col_ind, const *hipsparseMatDescr_t* bsr_descr, float *bsr_val, int *bsr_row_ptr, int *bsr_col_ind, int row_block_dim, int col_block_dim, void *p_buffer)

hipsparseStatus_t **hipsparseDcsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const double *csr_val, const int *csr_row_ptr, const int *csr_col_ind, const *hipsparseMatDescr_t* bsr_descr, double *bsr_val, int *bsr_row_ptr, int *bsr_col_ind, int row_block_dim, int col_block_dim, void *p_buffer)

hipsparseStatus_t **hipsparseCcsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const hipComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, const *hipsparseMatDescr_t* bsr_descr, hipComplex *bsr_val, int *bsr_row_ptr, int *bsr_col_ind, int row_block_dim, int col_block_dim, void *p_buffer)

hipsparseStatus_t **hipsparseZcsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dir, int m, int n, const *hipsparseMatDescr_t* csr_descr, const hipDoubleComplex *csr_val, const int *csr_row_ptr, const int *csr_col_ind, const *hipsparseMatDescr_t* bsr_descr, hipDoubleComplex *bsr_val, int *bsr_row_ptr, int *bsr_col_ind, int row_block_dim, int col_block_dim, void *p_buffer)

Convert a sparse CSR matrix into a sparse GEneral BSR matrix.

`hipsparseXcsr2gebsr` converts a CSR matrix into a GEneral BSR matrix. It is assumed, that `bsr_val`, `bsr_col_ind` and `bsr_row_ptr` are allocated. Allocation size for `bsr_row_ptr` is computed as `mb+1` where `mb` is the number of block rows in the GEneral BSR matrix. Allocation size for `bsr_val` and `bsr_col_ind` is computed using `csr2gebsr_nnz()` which also fills in `bsr_row_ptr`.

1.13.26 hipsparseXbsr2csr()

hipsparseStatus_t **hipsparseSbsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, float *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseDbsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, double *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseCbsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, hipComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseZbsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int blockDim, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

Convert a sparse BSR matrix into a sparse CSR matrix.

`hipsparseXbsr2csr` converts a BSR matrix into a CSR matrix. It is assumed, that `csr_val`, `csr_col_ind` and `csr_row_ptr` are allocated. Allocation size for `csr_row_ptr` is computed by the number of block rows multiplied by the block dimension plus one. Allocation for `csr_val` and `csr_col_ind` is computed by the the number of blocks in the BSR matrix multiplied by the block dimension squared.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.27 hipsparseXgebsr2csr()

hipsparseStatus_t **hipsparseSgebsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDim, int colBlockDim, const *hipsparseMatDescr_t* descrC, float *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseDgebsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDim, int colBlockDim, const *hipsparseMatDescr_t* descrC, double *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseCgebsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDim, int colBlockDim, const *hipsparseMatDescr_t* descrC, hipComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

hipsparseStatus_t **hipsparseZgebsr2csr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDim, int colBlockDim, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *csrValC, int *csrRowPtrC, int *csrColIndC)

Convert a sparse general BSR matrix into a sparse CSR matrix.

`hipsparseXgebsr2csr` converts a BSR matrix into a CSR matrix. It is assumed, that `csr_val`, `csr_col_ind` and `csr_row_ptr` are allocated. Allocation size for `csr_row_ptr` is computed by the number of block rows multiplied by the block dimension plus one. Allocation for `csr_val` and `csr_col_ind` is computed by the the number of blocks in the BSR matrix multiplied by the product of the block dimensions.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.28 hipsparseXcsr2csr_compress()

hipsparseStatus_t **hipsparseScsr2csr_compress**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrColIndA, const int *csrRowPtrA, int nnzA, const int *nnzPerRow, float *csrValC, int *csrColIndC, int *csrRowPtrC, float tol)

hipsparseStatus_t **hipsparseDcsr2csr_compress**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrColIndA, const int *csrRowPtrA, int nnzA, const int *nnzPerRow, double *csrValC, int *csrColIndC, int *csrRowPtrC, double tol)

hipsparseStatus_t **hipsparseCcsr2csr_compress**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const hipComplex *csrValA, const int *csrColIndA, const int *csrRowPtrA, int nnzA, const int *nnzPerRow, hipComplex *csrValC, int *csrColIndC, int *csrRowPtrC, hipComplex tol)

hipsparseStatus_t **hipsparseZcsr2csr_compress**(*hipsparseHandle_t* handle, int m, int n, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrValA, const int *csrColIndA, const int *csrRowPtrA, int nnzA, const int *nnzPerRow, hipDoubleComplex *csrValC, int *csrColIndC, int *csrRowPtrC, hipDoubleComplex tol)

Convert a sparse CSR matrix into a compressed sparse CSR matrix.

`hipsparseXcsr2csr_compress` converts a CSR matrix into a compressed CSR matrix by removing entries in the input CSR matrix that are below a non-negative threshold `tol`

Note: In the case of complex matrices only the magnitude of the real part of `tol` is used.

1.13.29 hipsparseXpruneCsr2csr_bufferSize()

hipsparseStatus_t **hipsparseSpruneCsr2csr_bufferSize**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *threshold, const *hipsparseMatDescr_t* descrC, const float *csrValC, const int *csrRowPtrC, const int *csrColIndC, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneCsr2csr_bufferSize**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *threshold, const *hipsparseMatDescr_t* descrC, const double *csrValC, const int *csrRowPtrC, const int *csrColIndC, size_t *bufferSize)

Convert and prune sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csr_bufferSize` returns the size of the temporary buffer that is required by `hipsparseXpruneCsr2csrNnz` and hipsparseXpruneCsr2csr. The temporary storage buffer must be allocated by the user.

1.13.30 hipsparseXpruneCsr2csr_bufferSizeExt()

hipsparseStatus_t **hipsparseSpruneCsr2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *threshold, const *hipsparseMatDescr_t* descrC, const float *csrValC, const int *csrRowPtrC, const int *csrColIndC, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneCsr2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *threshold, const *hipsparseMatDescr_t* descrC, const double *csrValC, const int *csrRowPtrC, const int *csrColIndC, size_t *bufferSize)

Convert and prune sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csr_bufferSizeExt` returns the size of the temporary buffer that is required by `hipsparseXpruneCsr2csrNnz` and hipsparseXpruneCsr2csr. The temporary storage buffer must be allocated by the user.

1.13.31 hipsparseXpruneCsr2csrNnz()

hipsparseStatus_t **hipsparseSpruneCsr2csrNnz**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *threshold, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr, void *buffer)

hipsparseStatus_t **hipsparseDpruneCsr2csrNnz**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *threshold, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr, void *buffer)

Convert and prune sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csrNnz` computes the number of nonzero elements per row and the total number of nonzero elements in a sparse CSR matrix once elements less than the threshold are pruned from the matrix.

Note: The routine does support asynchronous execution if the pointer mode is set to device.

1.13.32 hipsparseXpruneCsr2csr()

hipsparseStatus_t **hipsparseSpruneCsr2csr**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *threshold, const *hipsparseMatDescr_t* descrC, float *csrValC, const int *csrRowPtrC, int *csrColIndC, void *buffer)

hipsparseStatus_t **hipsparseDpruneCsr2csr**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *threshold, const *hipsparseMatDescr_t* descrC, double *csrValC, const int *csrRowPtrC, int *csrColIndC, void *buffer)

Convert and prune sparse CSR matrix into a sparse CSR matrix.

This function converts the sparse CSR matrix A into a sparse CSR matrix C by pruning values in A that are less than the threshold. All the parameters are assumed to have been pre-allocated by the user. The user first calls hipsparseXpruneCsr2csr_bufferSize() to determine the size of the buffer used by hipsparseXpruneCsr2csrNnz() and hipsparseXpruneCsr2csr() which the user then allocates. The user then allocates `csr_row_ptr_C` to have `m+1` elements and then calls hipsparseXpruneCsr2csrNnz() which fills in the `csr_row_ptr_C` array stores then number of elements that are larger than the pruning threshold in `nnz_total_dev_host_ptr`. The user then calls hipsparseXpruneCsr2csr() to complete the conversion. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.33 hipsparseXpruneCsr2csrByPercentage_bufferSize()

hipsparseStatus_t **hipsparseSpruneCsr2csrByPercentage_bufferSize**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, float percentage, const *hipsparseMatDescr_t* descrC, const float *csrValC, const int *csrRowPtrC, const int *csrColIndC, *pruneInfo_t* info, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneCsr2csrByPercentage_bufferSize**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, double percentage, const *hipsparseMatDescr_t* descrC, const double *csrValC, const int *csrRowPtrC, const int *csrColIndC, *pruneInfo_t* info, size_t *bufferSize)

Convert and prune by percentage a sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csrByPercentage_bufferSize` returns the size of the temporary buffer that is required by `hipsparseXpruneCsr2csrNnzByPercentage`. The temporary storage buffer must be allocated by the user.

1.13.34 hipsparseXpruneCsr2csrByPercentage_bufferSizeExt()

hipsparseStatus_t **hipsparseSpruneCsr2csrByPercentage_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, float percentage, const *hipsparseMatDescr_t* descrC, const float *csrValC, const int *csrRowPtrC, const int *csrColIndC, *pruneInfo_t* info, size_t *bufferSize)

hipsparseStatus_t **hipsparseDpruneCsr2csrByPercentage_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, double percentage, const *hipsparseMatDescr_t* descrC, const double *csrValC, const int *csrRowPtrC, const int *csrColIndC, *pruneInfo_t* info, size_t *bufferSize)

Convert and prune by percentage a sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csrByPercentage_bufferSizeExt` returns the size of the temporary buffer that is required by `hipsparseXpruneCsr2csrNnzByPercentage`. The temporary storage buffer must be allocated by the user.

1.13.35 hipsparseXpruneCsr2csrNnzByPercentage()

hipsparseStatus_t **hipsparseSpruneCsr2csrNnzByPercentage**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, float percentage, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr, *pruneInfo_t* info, void *buffer)

hipsparseStatus_t **hipsparseDpruneCsr2csrNnzByPercentage**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, double percentage, const *hipsparseMatDescr_t* descrC, int *csrRowPtrC, int *nnzTotalDevHostPtr, *pruneInfo_t* info, void *buffer)

Convert and prune by percentage a sparse CSR matrix into a sparse CSR matrix.

`hipsparseXpruneCsr2csrNnzByPercentage` computes the number of nonzero elements per row and the total number of nonzero elements in a sparse CSR matrix once elements less than the threshold are pruned from the matrix.

Note: The routine does support asynchronous execution if the pointer mode is set to device.

1.13.36 hipsparseXpruneCsr2csrByPercentage()

hipsparseStatus_t **hipsparseSpruneCsr2csrByPercentage**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, float percentage, const *hipsparseMatDescr_t* descrC, float *csrValC, const int *csrRowPtrC, int *csrColIndC, *pruneInfo_t* info, void *buffer)

hipsparseStatus_t **hipsparseDpruneCsr2csrByPercentage**(*hipsparseHandle_t* handle, int m, int n, int nnzA, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, double percentage, const *hipsparseMatDescr_t* descrC, double *csrValC, const int *csrRowPtrC, int *csrColIndC, *pruneInfo_t* info, void *buffer)

Convert and prune by percentage a sparse CSR matrix into a sparse CSR matrix.

This function converts the sparse CSR matrix A into a sparse CSR matrix C by pruning values in A that are less than the threshold. All the parameters are assumed to have been pre-allocated by the user. The user first calls hipsparseXpruneCsr2csr_bufferSize() to determine the size of the buffer used by hipsparseXpruneCsr2csrNnz() and hipsparseXpruneCsr2csr() which the user then allocates. The user then allocates `csr_row_ptr_C` to have `m+1` elements and then calls hipsparseXpruneCsr2csrNnz() which fills in the `csr_row_ptr_C` array stores then number of elements that are larger than the pruning threshold in `nnz_total_dev_host_ptr`. The user then calls hipsparseXpruneCsr2csr() to complete the conversion. It is executed asynchronously with respect to the host and may return control to the application on the host before the entire result is ready.

1.13.37 hipsparseXhyb2csr()

hipsparseStatus_t **hipsparseShyb2csr**(*hipsparseHandle_t* handle, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, float *csrSortedValA, int *csrSortedRowPtrA, int *csrSortedColIndA)

hipsparseStatus_t **hipsparseDhyb2csr**(*hipsparseHandle_t* handle, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, double *csrSortedValA, int *csrSortedRowPtrA, int *csrSortedColIndA)

hipsparseStatus_t **hipsparseChyb2csr**(*hipsparseHandle_t* handle, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, hipComplex *csrSortedValA, int *csrSortedRowPtrA, int *csrSortedColIndA)

hipsparseStatus_t **hipsparseZhyb2csr**(*hipsparseHandle_t* handle, const *hipsparseMatDescr_t* descrA, const *hipsparseHybMat_t* hybA, hipDoubleComplex *csrSortedValA, int *csrSortedRowPtrA, int *csrSortedColIndA)

Convert a sparse HYB matrix into a sparse CSR matrix.

`hipsparseXhyb2csr` converts a HYB matrix into a CSR matrix.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.38 hipsparseXcoo2csr()

hipsparseStatus_t **hipsparseXcoo2csr**(*hipsparseHandle_t* handle, const int *cooRowInd, int nnz, int m, int *csrRowPtr, *hipsparseIndexBase_t* idxBase)

Convert a sparse COO matrix into a sparse CSR matrix.

`hipsparseXcoo2csr` converts the COO array containing the row indices into a CSR array of row offsets, that point to the start of every row. It is assumed that the COO row index array is sorted.

Note: It can also be used, to convert a COO array containing the column indices into a CSC array of column offsets, that point to the start of every column. Then, it is assumed that the COO column index array is sorted, instead.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.39 hipsparseCreateIdentityPermutation()

hipsparseStatus_t **hipsparseCreateIdentityPermutation**(*hipsparseHandle_t* handle, int n, int *p)

Create the identity map.

`hipsparseCreateIdentityPermutation` stores the identity map in `p`, such that $p = 0 : 1 : (n - 1)$.

```
for(i = 0; i < n; ++i)
{
    p[i] = i;
}
```

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.40 hipsparseXcsrsort_bufferSizeExt()

hipsparseStatus_t **hipsparseXcsrsort_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, const int *csrRowPtr, const int *csrColInd, size_t *pBufferSizeInBytes)

Sort a sparse CSR matrix.

`hipsparseXcsrsort_bufferSizeExt` returns the size of the temporary storage buffer required by *hipsparseXcsrsort()*. The temporary storage buffer must be allocated by the user.

1.13.41 hipsparseXcsrsort()

hipsparseStatus_t **hipsparseXcsrsort**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, const int *csrRowPtr, int *csrColInd, int *P, void *pBuffer)

Sort a sparse CSR matrix.

`hipsparseXcsrsort` sorts a matrix in CSR format. The sorted permutation vector `perm` can be used to obtain sorted `csr_val` array. In this case, `perm` must be initialized as the identity permutation, see *hipsparseCreateIdentityPermutation()*.

`hipsparseXcsrsort` requires extra temporary storage buffer that has to be allocated by the user. Storage buffer size can be determined by *hipsparseXcsrsort_bufferSizeExt()*.

Note: `perm` can be `NULL` if a sorted permutation vector is not required.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.42 hipsparseXcscsort_bufferSizeExt()

hipsparseStatus_t **hipsparseXcscsort_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, const int *cscColPtr, const int *cscRowInd, size_t *pBufferSizeInBytes)

Sort a sparse CSC matrix.

`hipsparseXcscsort_bufferSizeExt` returns the size of the temporary storage buffer required by *hipsparseXcscsort()*. The temporary storage buffer must be allocated by the user.

1.13.43 hipsparseXcscsort()

hipsparseStatus_t **hipsparseXcscsort**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, const int *cscColPtr, int *cscRowInd, int *P, void *pBuffer)

Sort a sparse CSC matrix.

`hipsparseXcscsort` sorts a matrix in CSC format. The sorted permutation vector `perm` can be used to obtain sorted `csc_val` array. In this case, `perm` must be initialized as the identity permutation, see *hipsparseCreateIdentityPermutation()*.

`hipsparseXcscsort` requires extra temporary storage buffer that has to be allocated by the user. Storage buffer size can be determined by *hipsparseXcscsort_bufferSizeExt()*.

Note: `perm` can be `NULL` if a sorted permutation vector is not required.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.44 hipsparseXcoosort_bufferSizeExt()

hipsparseStatus_t **hipsparseXcoosort_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, const int *cooRows, const int *cooCols, size_t *pBufferSizeInBytes)

Sort a sparse COO matrix.

`hipsparseXcoosort_bufferSizeExt` returns the size of the temporary storage buffer required by hipsparseXcoosort(). The temporary storage buffer must be allocated by the user.

1.13.45 hipsparseXcoosortByRow()

hipsparseStatus_t **hipsparseXcoosortByRow**(*hipsparseHandle_t* handle, int m, int n, int nnz, int *cooRows, int *cooCols, int *P, void *pBuffer)

Sort a sparse COO matrix by row.

`hipsparseXcoosortByRow` sorts a matrix in COO format by row. The sorted permutation vector `perm` can be used to obtain sorted `coo_val` array. In this case, `perm` must be initialized as the identity permutation, see *hipsparseCreateIdentityPermutation()*.

`hipsparseXcoosortByRow` requires extra temporary storage buffer that has to be allocated by the user. Storage buffer size can be determined by *hipsparseXcoosort_bufferSizeExt()*.

Note: `perm` can be `NULL` if a sorted permutation vector is not required.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.46 hipsparseXcoosortByColumn()

hipsparseStatus_t **hipsparseXcoosortByColumn**(*hipsparseHandle_t* handle, int m, int n, int nnz, int *cooRows, int *cooCols, int *P, void *pBuffer)

Sort a sparse COO matrix by column.

`hipsparseXcoosortByColumn` sorts a matrix in COO format by column. The sorted permutation vector `perm` can be used to obtain sorted `coo_val` array. In this case, `perm` must be initialized as the identity permutation, see *hipsparseCreateIdentityPermutation()*.

`hipsparseXcoosortByColumn` requires extra temporary storage buffer that has to be allocated by the user. Storage buffer size can be determined by *hipsparseXcoosort_bufferSizeExt()*.

Note: `perm` can be `NULL` if a sorted permutation vector is not required.

Note: This function is non blocking and executed asynchronously with respect to the host. It may return before the actual computation has finished.

1.13.47 hipsparseXgebsr2gebsr_bufferSize()

hipsparseStatus_t **hipsparseSgebsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, int rowBlockDimC, int colBlockDimC, int *bufferSize)

hipsparseStatus_t **hipsparseDgebsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, int rowBlockDimC, int colBlockDimC, int *bufferSize)

hipsparseStatus_t **hipsparseCgebsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, int rowBlockDimC, int colBlockDimC, int *bufferSize)

hipsparseStatus_t **hipsparseZgebsr2gebsr_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, int rowBlockDimC, int colBlockDimC, int *bufferSize)

This function computes the the size of the user allocated temporary storage buffer used when converting a sparse general BSR matrix to another sparse general BSR matrix.

`hipsparseXgebsr2gebsr_bufferSize` returns the size of the temporary storage buffer that is required by *hipsparseXgebsr2gebsrNnz()* and hipsparseXgebsr2gebsr(). The temporary storage buffer must be allocated by the user.

1.13.48 hipsparseXgebsr2gebsrNnz()

hipsparseStatus_t **hipsparseXgebsr2gebsrNnz**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, const *hipsparseMatDescr_t* descrC, int *bsrRowPtrC, int rowBlockDimC, int colBlockDimC, int *nnzTotalDevHostPtr, void *buffer)

This function is used when converting a general BSR sparse matrix `A` to another general BSR sparse matrix `C`. Specifically, this function determines the number of non-zero blocks that will exist in `C` (stored using either a host or device pointer), and computes the row pointer array for `C`.

The routine does support asynchronous execution.

1.13.49 hipsparseXgebsr2gebsr()

hipsparseStatus_t **hipsparseSgebsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const float *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, const *hipsparseMatDescr_t* descrC, float *bsrValC, int *bsrRowPtrC, int *bsrColIndC, int rowBlockDimC, int colBlockDimC, void *buffer)

hipsparseStatus_t **hipsparseDgebsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const double *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, const *hipsparseMatDescr_t* descrC, double *bsrValC, int *bsrRowPtrC, int *bsrColIndC, int rowBlockDimC, int colBlockDimC, void *buffer)

hipsparseStatus_t **hipsparseCgebsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, const *hipsparseMatDescr_t* descrC, hipComplex *bsrValC, int *bsrRowPtrC, int *bsrColIndC, int rowBlockDimC, int colBlockDimC, void *buffer)

hipsparseStatus_t **hipsparseZgebsr2gebsr**(*hipsparseHandle_t* handle, *hipsparseDirection_t* dirA, int mb, int nb, int nnzb, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *bsrValA, const int *bsrRowPtrA, const int *bsrColIndA, int rowBlockDimA, int colBlockDimA, const *hipsparseMatDescr_t* descrC, hipDoubleComplex *bsrValC, int *bsrRowPtrC, int *bsrColIndC, int rowBlockDimC, int colBlockDimC, void *buffer)

This function converts the general BSR sparse matrix `A` to another general BSR sparse matrix `C`.

The conversion uses three steps. First, the user calls hipsparseXgebsr2gebsr_bufferSize() to determine the size of the required temporary storage buffer. The user then allocates this buffer. Secondly, the user then allocates `mb_C+1` integers for the row pointer array for `C` where `mb_C`=(m+row_block_dim_C-1)/row_block_dim_C. The user then calls *hipsparseXgebsr2gebsrNnz()* to fill in the row pointer array for `C` (`bsr_row_ptr_C`) and determine the number of non-zero blocks that will exist in `C`. Finally, the user allocates space for the colimn indices array of `C` to have `nnzb_C` elements and space for the values array of `C` to have `nnzb_C*roc_block_dim_C*col_block_dim_C` and then calls hipsparseXgebsr2gebsr() to complete the conversion.

1.13.50 hipsparseXcsru2csr_bufferSizeExt()

hipsparseStatus_t **hipsparseScsru2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, float *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseDcsru2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, double *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseCcsru2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, hipComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, size_t *pBufferSizeInBytes)

hipsparseStatus_t **hipsparseZcsru2csr_bufferSizeExt**(*hipsparseHandle_t* handle, int m, int n, int nnz, hipDoubleComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, size_t *pBufferSizeInBytes)

This function calculates the amount of temporary storage required for hipsparseXcsru2csr() and hipsparseXcsr2csru().

1.13.51 hipsparseXcsru2csr()

hipsparseStatus_t **hipsparseScsru2csr**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, float *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseDcsru2csr**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, double *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseCcsru2csr**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseZcsru2csr**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

This function converts unsorted CSR format to sorted CSR format. The required temporary storage has to be allocated by the user.

1.13.52 hipsparseXcsr2csru()

hipsparseStatus_t **hipsparseScsr2csru**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, float *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseDcsr2csru**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, double *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseCcsr2csru**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, hipComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

hipsparseStatus_t **hipsparseZcsr2csru**(*hipsparseHandle_t* handle, int m, int n, int nnz, const *hipsparseMatDescr_t* descrA, hipDoubleComplex *csrVal, const int *csrRowPtr, int *csrColInd, *csru2csrInfo_t* info, void *pBuffer)

This function converts sorted CSR format to unsorted CSR format. The required temporary storage has to be allocated by the user.

1.14 Sparse Reordering Functions

This module holds all sparse reordering routines.

1.14.1 hipsparseXcsrcolor()

hipsparseStatus_t **hipsparseScsrcolor**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const float *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *fractionToColor, int *ncolors, int *coloring, int *reordering, *hipsparseColorInfo_t* info)

hipsparseStatus_t **hipsparseDcsrcolor**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *fractionToColor, int *ncolors, int *coloring, int *reordering, *hipsparseColorInfo_t* info)

hipsparseStatus_t **hipsparseCcsrcolor**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const float *fractionToColor, int *ncolors, int *coloring, int *reordering, *hipsparseColorInfo_t* info)

hipsparseStatus_t **hipsparseZcsrcolor**(*hipsparseHandle_t* handle, int m, int nnz, const *hipsparseMatDescr_t* descrA, const hipDoubleComplex *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *fractionToColor, int *ncolors, int *coloring, int *reordering, *hipsparseColorInfo_t* info)

Coloring of the adjacency graph of the matrix A stored in the CSR format.

`hipsparseXcsrcolor` performs the coloring of the undirected graph represented by the (symmetric) sparsity pattern of the matrix A stored in CSR format. Graph coloring is a way of coloring the nodes of a graph such that no two adjacent nodes are of the same color. The `fraction_to_color` is a parameter to only color a given percentage of the graph nodes, the remaining uncolored nodes receive distinct new colors. The optional `reordering` array is a permutation array such that unknowns of the same color are grouped. The matrix A must be stored as a general matrix with a symmetric sparsity pattern, and if the matrix A is non-symmetric then the user is responsible to provide the symmetric part $\frac{A+A^T}{2}$.

1.15 Sparse Generic Functions

This module holds all sparse generic routines.

The sparse generic routines describe operations that manipulate sparse matrices.

1.15.1 hipsparseAxpby()

hipsparseStatus_t **hipsparseAxpby**(*hipsparseHandle_t* handle, const void *alpha, *hipsparseSpVecDescr_t* vecX, const void *beta, *hipsparseDnVecDescr_t* vecY)

1.15.2 hipsparseGather()

hipsparseStatus_t **hipsparseGather**(*hipsparseHandle_t* handle, *hipsparseDnVecDescr_t* vecY, *hipsparseSpVecDescr_t* vecX)

1.15.3 hipsparseScatter()

hipsparseStatus_t **hipsparseScatter**(*hipsparseHandle_t* handle, *hipsparseSpVecDescr_t* vecX, *hipsparseDnVecDescr_t* vecY)

1.15.4 hipsparseRot()

hipsparseStatus_t **hipsparseRot**(*hipsparseHandle_t* handle, const void *c_coeff, const void *s_coeff, *hipsparseSpVecDescr_t* vecX, *hipsparseDnVecDescr_t* vecY)

1.15.5 hipsparseSparseToDense_bufferSize()

hipsparseStatus_t **hipsparseSparseToDense_bufferSize**(*hipsparseHandle_t* handle, *hipsparseSpMatDescr_t* matA, *hipsparseDnMatDescr_t* matB, *hipsparseSparseToDenseAlg_t* alg, size_t *bufferSize)

1.15.6 hipsparseSparseToDense()

hipsparseStatus_t **hipsparseSparseToDense**(*hipsparseHandle_t* handle, *hipsparseSpMatDescr_t* matA, *hipsparseDnMatDescr_t* matB, *hipsparseSparseToDenseAlg_t* alg, void *externalBuffer)

1.15.7 hipsparseDenseToSparse_bufferSize()

hipsparseStatus_t **hipsparseDenseToSparse_bufferSize**(*hipsparseHandle_t* handle, *hipsparseDnMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, *hipsparseDenseToSparseAlg_t* alg, size_t *bufferSize)

1.15.8 hipsparseDenseToSparse_analysis()

hipsparseStatus_t **hipsparseDenseToSparse_analysis**(*hipsparseHandle_t* handle, *hipsparseDnMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, *hipsparseDenseToSparseAlg_t* alg, void *externalBuffer)

1.15.9 hipsparseDenseToSparse_convert()

hipsparseStatus_t **hipsparseDenseToSparse_convert**(*hipsparseHandle_t* handle, *hipsparseDnMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, *hipsparseDenseToSparseAlg_t* alg, void *externalBuffer)

1.15.10 hipsparseSpVV_bufferSize()

hipsparseStatus_t **hipsparseSpVV_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opX, *hipsparseSpVecDescr_t* vecX, *hipsparseDnVecDescr_t* vecY, void *result, hipDataType computeType, size_t *bufferSize)

1.15.11 hipsparseSpVV()

hipsparseStatus_t **hipsparseSpVV**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opX, *hipsparseSpVecDescr_t* vecX, *hipsparseDnVecDescr_t* vecY, void *result, hipDataType computeType, void *externalBuffer)

1.15.12 hipsparseSpMV_bufferSize()

hipsparseStatus_t **hipsparseSpMV_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnVecDescr_t* vecX, const void *beta, const *hipsparseDnVecDescr_t* vecY, hipDataType computeType, *hipsparseSpMVAlg_t* alg, size_t *bufferSize)

1.15.13 hipsparseSpMV()

hipsparseStatus_t **hipsparseSpMV**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnVecDescr_t* vecX, const void *beta, const *hipsparseDnVecDescr_t* vecY, hipDataType computeType, *hipsparseSpMVAlg_t* alg, void *externalBuffer)

1.15.14 hipsparseSpMM_bufferSize()

hipsparseStatus_t **hipsparseSpMM_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const void *beta, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpMMAlg_t* alg, size_t *bufferSize)

1.15.15 hipsparseSpMM_preprocess()

hipsparseStatus_t **hipsparseSpMM_preprocess**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const void *beta, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpMMAlg_t* alg, void *externalBuffer)

1.15.16 hipsparseSpMM()

hipsparseStatus_t **hipsparseSpMM**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const void *beta, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpMMAlg_t* alg, void *externalBuffer)

1.15.17 hipsparseSpGEMM_createDescr()

hipsparseStatus_t **hipsparseSpGEMM_createDescr**(*hipsparseSpGEMMDescr_t* *descr)

1.15.18 hipsparseSpGEMM_destroyDescr()

hipsparseStatus_t **hipsparseSpGEMM_destroyDescr**(*hipsparseSpGEMMDescr_t* descr)

1.15.19 hipsparseSpGEMM_workEstimation()

hipsparseStatus_t **hipsparseSpGEMM_workEstimation**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, *hipsparseSpMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, const void *beta, *hipsparseSpMatDescr_t* matC, hipDataType computeType, *hipsparseSpGEMMAlg_t* alg, *hipsparseSpGEMMDescr_t* spgemmDescr, size_t *bufferSize1, void *externalBuffer1)

1.15.20 hipsparseSpGEMM_compute()

hipsparseStatus_t **hipsparseSpGEMM_compute**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, *hipsparseSpMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, const void *beta, *hipsparseSpMatDescr_t* matC, hipDataType computeType, *hipsparseSpGEMMAlg_t* alg, *hipsparseSpGEMMDescr_t* spgemmDescr, size_t *bufferSize2, void *externalBuffer2)

1.15.21 hipsparseSpGEMM_copy()

hipsparseStatus_t **hipsparseSpGEMM_copy**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, *hipsparseSpMatDescr_t* matA, *hipsparseSpMatDescr_t* matB, const void *beta, *hipsparseSpMatDescr_t* matC, hipDataType computeType, *hipsparseSpGEMMAlg_t* alg, *hipsparseSpGEMMDescr_t* spgemmDescr)

1.15.22 hipsparseSDDMM_bufferSize()

hipsparseStatus_t **hipsparseSDDMM_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseDnMatDescr_t* A, const *hipsparseDnMatDescr_t* B, const void *beta, *hipsparseSpMatDescr_t* C, hipDataType computeType, *hipsparseSDDMMAlg_t* alg, size_t *bufferSize)

1.15.23 hipsparseSDDMM_preprocess()

hipsparseStatus_t **hipsparseSDDMM_preprocess**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseDnMatDescr_t* A, const *hipsparseDnMatDescr_t* B, const void *beta, *hipsparseSpMatDescr_t* C, hipDataType computeType, *hipsparseSDDMMAlg_t* alg, void *tempBuffer)

1.15.24 hipsparseSDDMM()

hipsparseStatus_t **hipsparseSDDMM**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseDnMatDescr_t* A, const *hipsparseDnMatDescr_t* B, const void *beta, *hipsparseSpMatDescr_t* C, hipDataType computeType, *hipsparseSDDMMAlg_t* alg, void *tempBuffer)

1.15.25 hipsparseSpSV_createDescr()

hipsparseStatus_t **hipsparseSpSV_createDescr**(*hipsparseSpSVDescr_t* *descr)

1.15.26 hipsparseSpSV_destroyDescr()

hipsparseStatus_t **hipsparseSpSV_destroyDescr**(*hipsparseSpSVDescr_t* descr)

1.15.27 hipsparseSpSV_bufferSize()

hipsparseStatus_t **hipsparseSpSV_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnVecDescr_t* x, const *hipsparseDnVecDescr_t* y, hipDataType computeType, *hipsparseSpSVAlg_t* alg, *hipsparseSpSVDescr_t* spsvDescr, size_t *bufferSize)

1.15.28 hipsparseSpSV_analysis()

hipsparseStatus_t **hipsparseSpSV_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnVecDescr_t* x, const *hipsparseDnVecDescr_t* y, hipDataType computeType, *hipsparseSpSVAlg_t* alg, *hipsparseSpSVDescr_t* spsvDescr, void *externalBuffer)

1.15.29 hipsparseSpSV_solve()

hipsparseStatus_t **hipsparseSpSV_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnVecDescr_t* x, const *hipsparseDnVecDescr_t* y, hipDataType computeType, *hipsparseSpSVAlg_t* alg, *hipsparseSpSVDescr_t* spsvDescr, void *externalBuffer)

1.15.30 hipsparseSpSM_createDescr()

hipsparseStatus_t **hipsparseSpSM_createDescr**(*hipsparseSpSMDescr_t* *descr)

1.15.31 hipsparseSpSM_destroyDescr()

hipsparseStatus_t **hipsparseSpSM_destroyDescr**(*hipsparseSpSMDescr_t* descr)

1.15.32 hipsparseSpSM_bufferSize()

hipsparseStatus_t **hipsparseSpSM_bufferSize**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpSMAlg_t* alg, *hipsparseSpSMDescr_t* spsmDescr, size_t *bufferSize)

1.15.33 hipsparseSpSM_analysis()

hipsparseStatus_t **hipsparseSpSM_analysis**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpSMAlg_t* alg, *hipsparseSpSMDescr_t* spsmDescr, void *externalBuffer)

1.15.34 hipsparseSpSM_solve()

hipsparseStatus_t **hipsparseSpSM_solve**(*hipsparseHandle_t* handle, *hipsparseOperation_t* opA, *hipsparseOperation_t* opB, const void *alpha, const *hipsparseSpMatDescr_t* matA, const *hipsparseDnMatDescr_t* matB, const *hipsparseDnMatDescr_t* matC, hipDataType computeType, *hipsparseSpSMAlg_t* alg, *hipsparseSpSMDescr_t* spsmDescr, void *externalBuffer)

INDEX

P